Eindhoven University of Technology

MASTER

An introduction to distributed computer systems : a study of Windows NT

Beens, E.

Award date:
1995

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit Eindhoven
Faculty of Electrical Engineering
Section of Digital Information Systems

Master's thesis:

An introduction to distributed computer systems

a study of Windows NT

E. Beens

Coaches	: ir. A.G.M. Geurts
	: ir. M.J.M van Weert
Supervisor	: prof. ir. M.P.J. Stevens
Period	: October1994 - August 1995

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses

Abstract

This thesis presents an introduction to the area of distributed computer systems. Nowadays many computers are connected by networks. An user operating such a computer is often very aware of the fact that it is connected to a network. Users have to know for instance addresses of other computers in order to gain access to data located there, or to execute a process there because that computer is better equipped. The ultimate goal of research in the area of distributed computer systems is to provide a view to the user of computers interconnected by a network, as if it were a single multiprocessor computer.

Basically there are four elements that can be distributed in a network; hardware, processing, data and control. This is not enough to describe a distributed computer system. The computers also have to cooperate according to some network wide strategy, to present the users a view of the network as one transparent functioning whole. A general definition of a distributed computer system is "a set of communicating processes, usually located at different nodes, cooperating to solve a common problem". This general definition is refined to a working definition, summarizing the properties an ideal distributed computer system should possess. The three primary characteristics are: (i) multiple computers, interconnected by (ii) a network maintaining (iii) a shared state. The shared state property, which is less obvious, is divided into multiple sub-properties. This working definition can serve as a starting-point for further discussion and as a reference for the study of distributed computer systems.

Some fundamental problems encountered in distributed computer systems and some techniques to overcome them are discussed. Especially the fact that the computers must communicate via messages over a network that can delay or loose these messages makes the implementation of a distributed computer system a greater challenge. It is discussed how to obtain a consistent global view of the distributed system, preserving causal relations between different processes. The remote procedure call is examined as a mechanism to distribute processes across the network. Furthermore it is discussed how replicas of data items can be managed so that if users access replicas of the same data item they all obtain consistent values. If users want to use the same data item at the same time, some access control in the form of transaction management is required. In order to be able to achieve system transparency, the topic of global naming is also examined. Finally we will discuss how a global file system could be created by making use of the earlier discussed issues.

The thesis will be concluded by a study of the operating system Windows NT in order to see if it contains distributed characteristics, and if it may be called a distributed operating system according to the presented working definition. Main conclusion of this study is that although it posses some distributed properties, it is not yet a truly distributed computer operating system. Because of the way in which Windows NT is structured, it does however provide a possibility to build a distributed computing environment on top of it. So although Windows NT is heading for the right direction, a long road still lays ahead.

Table of contents

Part I

In this part an introduction will be given to the world of distributed computer systems. The advantages of distributed computing will be pointed out. We will also discuss the characteristics we would like to see in an ideal distributed computer system, which will lead to a definition of a distributed computer system. This definition is not a strict one, but it will help us determine to what extent a computer system is distributed.

1. Introduction

The fact that computers can be linked together to form a network is of course common knowledge. A user of such a network however must be very aware of that. From the moment he sits down behind a computer that is connected to a network he is constantly reminded of that fact. He has to know for example the address or name of his own computer in order to log in from another computer to get to his personal data. Another problem arises when the user has an application that he wants to execute on another computer that is better equipped then his own. In that case the user himself must have the address of that computer, and has to make sure that his request for service gets to that particular computer.

Would it not be nice if a network of interconnected computers would feel and look like a single big multiprocessor computer to the user? Stronger even, that every computer from the network that the user uses feels and looks like his own. This would for instance mean that a user could log in at another computer than his own, but still see his own data as if he were sitting behind his own computer. This data is thus accessed from the remote computer at a location the user does not need to know.

The other problem mentioned above is of an user starting a process on his computer that is either too busy or does not have the proper hardware or software. Now suppose that the operating system looks for another computer that is fit for the job and uses it. The user does not need to know the address of that computer and even should not be aware of the fact that his process is running on a remote computer.

This is where the theory of distributed computer systems comes in the picture. In a distributed network there are four physical parts that could be distributed. These are: hardware, processing, data and control. A distributed computer system can be seen as a collection interconnected computers presenting itself to the users as one single multiprocessor computer. This means that a distributed computer system needs to hide the physical locations and addresses from the user, also called location transparency. This may have lead to the definition of a distributed system by Leslie Lamport: 'You know you have one when the crash of a computer you've never heard of stops you from getting any work done.'. In such a multiprocessor environment it can also happen that two users simultaneously call upon the same service, data item etc. Hence access must also be controlled to prevent inconsistencies.

Distributed computer systems offer of course some advantages. By copying and distributing data and services across the network, they can be brought closer to the user and thus increase availability. Having copies, or replicas, also provides better reliability and fault tolerance. If something breaks down, its tasks can be taken over by a replica. Furthermore does the multiprocessor environment of a distributed system have a potential performance enhancement by performing parallel processing.

Is this Utopia, or could a truly distributed computer system be developed? The section of Digital Information Systems of the university of technology Eindhoven has stated in its mission, that it wants to perform research in the area of distributed computer- and telecommunication systems in the future. This thesis is meant as an introduction to distributed computer systems for students and graduates who are going to perform further research in this area.

Here in part one, a general description of distributed computer systems and their properties will be given. Also a definition of distributed computer systems will be presented. In part two some fundamental problems will be discussed, and some techniques or approaches will be given to overcome these problems and obtain the desired characteristics of a distributed computer system. Finally in part three a commercially available operating system will be examined, to determine to what extent it is distributed. The examined operating system is Windows NT. Part one and two of this thesis are mainly based upon the books [mul93] and [cou94]. The book [gos91] also discusses distributed operating systems much in the same way, but was only found near the end of the thesis. It could however proof useful for further examination in the area of distributed computing.

2. distributed computer systems

A distributed computer system potentially provides significant advantages. One can for instance achieve performance enhancement by distributing processing over multiple processors in the network, and at the same time avoid contention and bottlenecks that exist in single uniprocessor and multiprocessor computers. By distributing and replicating data across the network to the places it is most needed, it is possible to gain access to the data more efficiently. Reliability can be improved by distributing and replicating control and data over the network. That way when a process fails on a computer, the chance that all control over that process and data that is needed is lost, is very small. By continuing the process on another computer the problem can be solved. This is also why independent failure of computers can more easily be accomplished in a distributed computer system. In a distributed computer system it should also be possible to share hardware and software resources in a cost effective manner increasing productivity and lowering costs. Another advantage is that a distributed computer system tries to provide transparency to the user. The network beneath the distributed system is hidden from the user. Processes are started on remote locations without the user noticing it. Replicas of data are managed with no interference of the user. Data is presented to the user without him knowing the exact physical location.

As a consequence of the above, problems immediately start to rise. Some way has to be found to achieve this location transparency. Furthermore users may want to access the same data item at the same time, or replica's are updated independently by different users. Hence some access control is needed to deal with concurrency, and replicas need to be managed to stay consistent. Also the sharing of resources must be made possible. Concurrent processes have to be scheduled in an efficient and consistent way. Before starting to search for solutions for this kind of problems, let us first look closer into distributed systems in general.

If we only look at the word distributed, then there are four components in a computer system that can be distributed: hardware, processing, data and control. Only the fact that one or more of these components are distributed does not have to mean that the entire system is distributed. If we take for example the 'old fashion' network, hardware in the form of computers is distributed. If processing is not distributed over the network, and the network is only used to exchange messages or files, we can't possibly call the entire system distributed. A definition based solely upon the distribution of some of these component is therefore not sufficient. We must also say something about the way the distributed components interact.

We will look at a distributed computer system as the sum total of the computer network and the operating system that is distributed over the computers in the network. In fact it is the distributed operating system that makes a computer system distributed. So whenever we talk about a distributed computer system this is equivalent to a distributed operating system. The individual copies of the operating system at every computer are of course not able to provide the desired distributed characteristics by them selves. To provide the distributed services they have to cooperate and communicate with each other. This is done according to some networkwide known strategy which results in algorithms that are supported networkwide. These strategies and algorithms are implemented in the operating system, so the distributed parts of the system can work together towards the common goal. We will describe this as: “the different computers in the distributed system have to cooperate to maintain some shared state”.

Let us now look at how a distributed computer system could be structured. At every computer in the system an operating system kernel is running. This is not necessarily the same kernel everywhere, as long as the same set of services is presented to every user by the entire operating system. The kernel provides the basic operating system services. The job of the kernel is to operate the local machine, thus the kernel itself is not distributed. This kernel is generally made as small as possible which explains the often used term ‘micro-kernel’. On top of this kernel there is a layer of operating system services that are not provided by the kernel. This is the layer where the networkwide strategies and algorithms are translated into the distributed services. This layer makes use of the low-level routines of the kernel. Services are often accessed using the client/server model. The client/server model is not only a well understood and much used programming method, it also has another advantage. Because the client makes a request to a service in stead of a particular server, a higher degree of transparency can be reached. Finally on top of the services layer are the user applications. This structure of a distributed computer system is depicted in Figure 1.

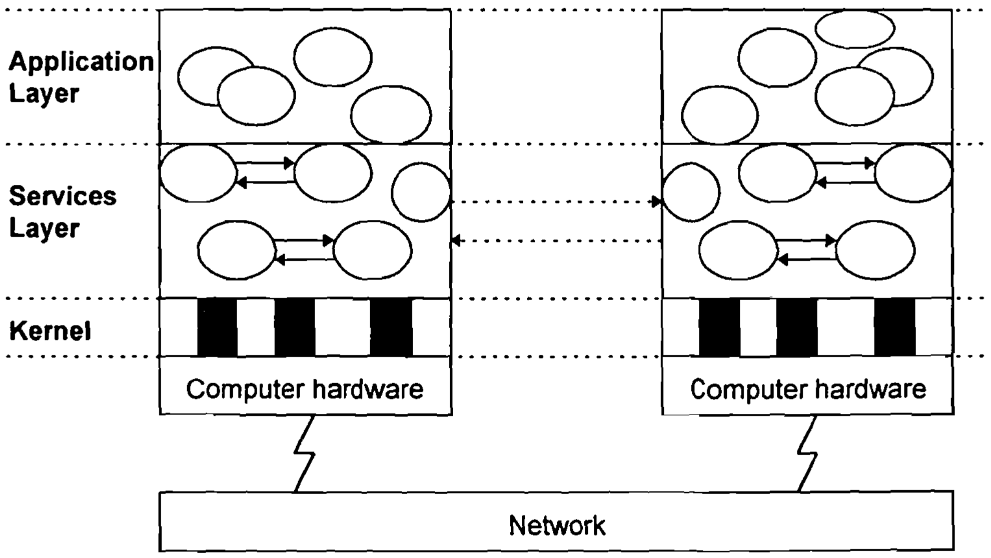


Figure 1 General structure of a distributed computer system

We can see a distributed computer system as an alternative approach to centralized computer systems and networking-based systems. The main difference between the distributed and the centralized approach is that, in the distributed case, processing and software can be transparently distributed over different nodes. The similarity is that both should look as though they were centralized, thus an user must have the feeling he sits behind a single computer. The main difference with a networked system is that the networked system does not appear as a single computer. For instance, an user explicitly has to transfer a file from one node to another. The similarity here is that both hardware and software are distributed.

In the next chapters we will present the properties we would like to see in an ideal distributed computer system, resulting in a working definition of distributed computer systems.

3. Primary characteristics

First we will try to give some primary characteristics a distributed computer system must posses. The most general description of a distributed computer system is: 'several computers doing something together'. A more accurate description would be: "a set of communicating processes, usually located at different nodes, cooperating to solve a common problem". From these descriptions and the discussion in chapter 2, we can conclude that a distributed system has the following primary characteristics:

1. Multiple Computers
2. Interconnections
3. Shared State

Multiple computers:
A distributed system contains more than one physical computer, each consisting of CPU('s), some local memory, possibly some stable storage like disks, and I/O paths to connect it with the environment. Here we make a distinction between multiple computers and multiple processors. In virtual form a single multiprocessor computer could be a distributed system. The processors each have their own local memory and they are connected to a kind of network e.g. the computer buss. The reason that we exclude these is because they in fact form a subset of distributed computer systems with more than one physical computer. The problems in multicomputer distributed systems are more challenging because of: greater delays, possible loss of messages, failure of interconnections, independent failure of computers etc. All the problems however that arise in multiprocessor computers can be dealt with, with the same techniques as in multicomputer distributed systems.

Interconnections:
The computers must be interconnected because if the computers can't talk to each other, then it is not going to be a very interesting distributed system. Building a system out of interconnected computers involves its own problems. The following issues must therefore be addressed:

- Independent failure - because there are several distinct computers involved, when one breaks down, others may keep going. We want the system to keep working after one or more have failed. The performance of course may get worse.

- Unreliable communication - one computer can not count on being able to communicate clearly with another computer, even if both are working properly. This is because interconnections between them are assumed to be not reliable. Messages may be corrupted due to noise, or connections may be unavailable because some student pulled the wrong plug.

- Unpredictable message arrival - Even if one computer can clearly communicate with the other, the exact time a message will arrive is not known due to delays in the network. Related messages can even be delivered out of order for instance because they followed another route through the network.
- Cost of communication - interconnections among computers have their own restrictions like higher latency and lower bandwidth. These must be taken in account when building a system of interconnected computers.
- Insecure communication - interconnections among computers are not always secure. Messages must be protected from eavesdropping. Furthermore messages may not be put on the lines by unauthorized people that may compromise the system.

The network characteristics mentioned above make the design of a distributed computer system more difficult. The problems that they might cause must be taken in account in the techniques that are used to achieve the distributed characteristics. Insecure communication is in practice of course an important issue in a network environment. It however involves its own specific set of techniques that can be implemented after a distributed computer system is built. We will concentrate ourselves upon achieving the distributed characteristics of a distributed computer system and will therefore not pay further attention to this issue in this thesis.

Shared state:
For an user a distributed computer system must look and feel like one single computer. The computers in the system have to work together to achieve this; they cooperate to maintain some shared state. Put in a more formal way, if the correct operation of the system is described in terms of some global invariants, then maintaining those invariants requires the correct and coordinated operation of multiple computers. As an example the computers in the system together should provide location transparency for the user. This means that the user does not have to know where his data, processes or services are located in the network. The computers could cooperate to implement some form of global naming of files in the system. Furthermore they could maintain distributed and replicated files in the network without the user knowing it by managing the distributed files and keeping replicated files up to date throughout the network.

This means we have to integrate the components of a distributed computer network to a functioning whole. Therefore we have to decide which techniques we are going to use network wide to achieve the distributed characteristics. These techniques must be incorporated in a distributed operating system, or each computer in the network at least has to know which technique or algorithm to use.

4. Definition of a distributed computer system

Now we have arrived at the point where we must give a definition of a distributed computer system so we can compare it with an implemented computer system in order to say if it is distributed or not. This is not as simple as it sounds. From the foregoing chapters it follows that there are many different properties that a distributed computer system should, or better said, could have. It is however not said that if a computer system misses one or two of these properties it is not distributed. The same thing goes the other way round, if a computer system possesses one of the properties it is not necessarily distributed. This is why we will give no exact definition of a distributed computer system. We will summarize properties that an ideal distributed computer system must have, and compare these with the properties of the examined computer system. We will say that when a system has a major portion of these properties it is distributed. Thus instead of an exact definition this will be a working definition.

We will base our working definition on the three primary characteristics we discovered in chapter 3:

- Multiple computers
- Interconnections
- Shared state

The two points stating that there must be multiple computers that are interconnected are obvious. In the design of practical distributed computer systems there are however some additional issues that should be addressed, although they are no strict requirements for the building of a distributed system. The multiple computers in computer networks are hardly ever exactly equal. This means that the distributed operating system must be able to deal with heterogeneity of hardware. It must be portable so to say. Another issue is that the amount of computers generally has the nasty habit of growing. So while designing a distributed system it must be kept in mind that they should be scaleable and extensible. Some of the issues concerning the interconnections between the computers, like for instance unreliable communication, have already been mentioned in a previous chapter.

The statement that these multiple computers must maintain some shared state is however rather general. Remember that we mean by this that the individual copies of the operating system at every computer have to communicate and cooperate to obtain the distributed characteristics, according to some networkwide strategies and algorithms. The shared state property consists of a set of sub-properties, and services the system must offer to achieve these. Therefore we will divide the shared state property into other sub-properties.

First of all location transparency must be offered to the users. Users must be able to access objects such as data, files and services without noticing if the objects are located at a remote location. This means that some form of global name service must exist. The same name is valid from everywhere in the system, and the user does not need to know the physical location.

Processing is one of the things that has to be distributed over the network in a transparent way. Different processes may however depend on each other. The outcome of one process might cause the execution of another one, or a process has to wait for the results of another process. These processes have a so called happened before relationship, while processes without a happened before relationship are said to be concurrent. This means that it must be possible to execute processes concurrently while preserving the happened before relationships. There has to exist a communication mechanism that allows access to remote services and execution of remote operations. A mechanism that is widely used to achieve this is the so called remote procedure call. It is thus a point of interest whether or not it is implemented in the distributed computer system.

Also data can be distributed across the network. The data is not only spread across the network but also copies, called replicas, are distributed to enhance availability and fault tolerance. The different replicas must be updated in so that the different users accessing it obtain a recent enough value that is consistent to possible updates at other locations. Data is usually stored in files. In order to allow transparent access to this files, the distributed computer system needs some global file server.

Because users may want to share data objects or system resources and these may be accessed by different users at the same time some access control has to be performed. This access control has to make sure that concurrent access is performed in a consistent way. Hence access control is a very important aspect of the topic distributed data above. Data is often accessed using a sequence of operations, called a transaction. A transaction can for instance update a certain data item based on reading other data items. If a single transaction is executed it will access data items in a consistent way. If more than one transaction accesses the same data items this is not necessarily so. Here distributed transaction management is needed.

The issues above are closely interrelated. The definition will be given as a summary of properties, and in this sense be a working definition. The definition consists of the three primary properties, where the shared state property is further refined. Note that, looking at the discussion above, these subproperties are closely interrelated. The first two primary characteristics are essential, while at the same time we would like to see as much subproperties of the shared state property as possible in a distributed system. The working definition in terms of basic characteristics and properties will be given down here. This working definition can serve as a starting-point for further discussion and as a reference for the study of distributed computer systems.

Working definition of a distributed computer system

1) Multiple computers
2) Interconnections
3) Shared state:
    - ♦ System transparency
        - Global naming
    - ♦ Distributed processing
        - Concurrent processing
        - Remote Procedure Calling
    - ♦ Distributed data
        - Replication management
        - Global file server
    - ♦ Access control
        - Sharing resources and data
        - Transaction management

There are some properties that we would like to see in a truly distributed computer system, but that do not directly belong to the working definition. Because of the sharing of resources and the possibilities of replicating services and data, fault tolerance and reliability can more easily be achieved in distributed systems. This is one of the most important advantages of distributed computer systems, and one of the reasons for the research in this area. Hence designers of distributed systems should take this into account. Other issues in the design of a computer system are scaleability and extensibility. When a computer system is developed it is important that is thought about how the system can be extended. This is of course important for every computer system, but because of the global characteristics of distributed computer systems it is especially important to look into how the distributed system could be extended without loosing its distributed characteristics.

Suggested literature

[cou94] - This book, together with [mul93], forms the foundation for part I and part II of this thesis. This book, however, takes a more practical approach to distributed systems.

[ens78] - In this article the author introduces distributed data processing systems. He also gives a definition of distributed systems in terms of a working definition. The characteristics of a distributed system are discussed and he tries to describe where the area of centralized systems stops and the area of distributed systems starts using some examples. The article served as the foundation for the definition presented in this thesis.

[joh94a] - An introduction to distributed systems. A general, layered structure of distributed operating systems is given. This layered structure is also presented in this thesis.

[joh94b] - A brief discussion of the benefits of distributed systems.

[mul87] - An introduction to distributed operating systems. Although it does not present an in-depth discussion, it helps to develop an understanding of distributed systems.

[mul93] - This book, together with [cou94], forms the foundation for part I and part II of this thesis. This book, however, takes a more formal approach to distributed systems.

[sta84] - Here the author gives an overview of the area of distributed computer systems. The characteristics of distributed systems are presented, together with a large amount of suggested literature. The characteristics mentioned are used in this thesis.

Part II

In this part we will discuss some fundamental problems in distributed computer systems and some techniques to overcome them. It is meant as an introduction to make the theory of distributed computer systems and their problems more understandable for people who are rather new to the subject. It is not the objective to present techniques and look if the distributed operating system examined later uses the same techniques.

5. Consistent global states in distributed systems

Like we have stated in our description of a distributed computer system, the computers in a distributed computer system have to maintain some shared state. It is therefore essential that a global state of the system can be detected. In a asynchronous distributed computer system this is not a trivial problem. A global view of the system can only be achieved by exchanging messages. Through loss and delay of messages the global state that is detected could be incomplete, too old or even inconsistent. In this chapter we will examine if in theory it is possible to attain a consistent global view of the system at each individual computer in the network. If this were not possible talking about distributed systems would be rather useless. Furthermore the discussion gives a good understanding of the problems involved in the management of processes that are executed simultaneously, but that may be involved in a happened before relationship to one another.

5.1 The model

The distributed computer system here is modeled in the following way: there are several processors that can communicate through exchange of messages. Programs can be divided into several processes that are distributed over the network. Now we want to know which processes run at a certain time. This could be useful for instance in case of monitoring of the processing, detecting deadlock or load balancing. In other words it is needed that a global state of the network can be detected.

Thus the distributed system is a collection of processes, and a network that provides communication channels between pairs of processes for message exchange. Each process can talk to all the other processes, if necessary through intermediary processes. Furthermore the distributed system is asynchronous and the network may deliver messages out of order. The individual interconnections between two machines however behave in a first-in-first-out fashion.

Each sequential process P_i consists of a sequence of k events e_i^k. A event may involve communication with another process. This communication is accomplished by the events *send(m)* and *receive(m)*, where m is the message identifier. The local history h_i of a process P_i can be represented by the sequence of events that occurred until that time, and the global history as the set of all these local histories. In an asynchronous distributed computer system there exists no global time frame, thus the events of the processes can only be ordered based on a 'cause and effect' relation. Two events are constrained to occur in a certain order if the first may affect the outcome of the second. This implies that information flows from the one to the other. Here this happens with the events in the same process, and between events in different processes that exchange *send* and *receive* messages. The causal relation can be defined in the following way:

1. If e_i^k and e_i^l are part of the local history h_i and $k<l$ then e_i^k causally precedes e_i^l.
2. If $e_i = send(m)$ and $e_j = receive(m)$ then e_i causally precedes e_j.
3. If e causally precedes e' and e' causally precedes e'', then e causally precedes e''.

If events in the global history are causally unrelated then these events are said to be concurrent, and may be executed simultaneously.

The above can be graphically represented by a space-time diagram (Figure 2). The horizontal lines represent the processes with their events. An arrow represents a message being sent from one process to another. It is now very easy to see which events are causally related; if a path can be traced from one event to another from left-to-right along the horizontal lines and the arrows, then they are causally related. Otherwise they are concurrent. So for instance in Figure 2 e_1^1 causally precedes e_2^2, but e_1^2 does not causally precede e_3^5.

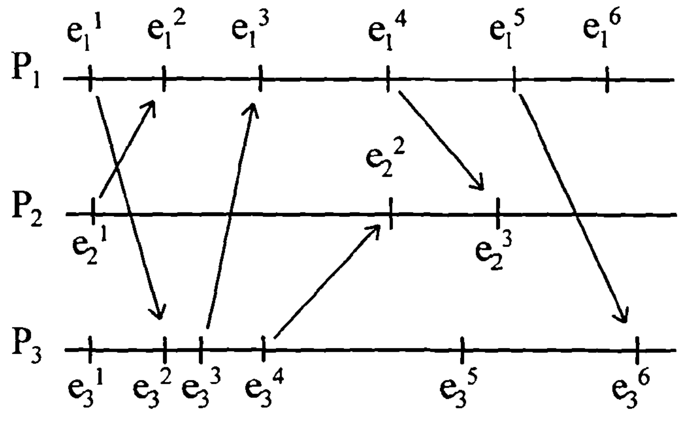
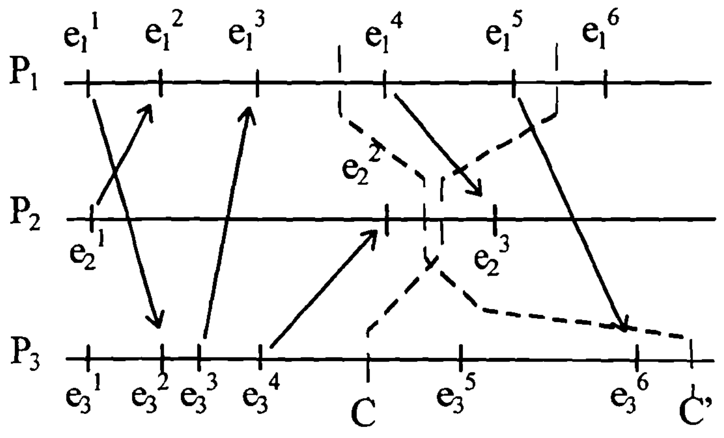


Figure 2 A time space diagram

5.2 Global state evaluation

We now want to say something about the global state of the system. Let σ_i^k denote the *local state* of process P_i after the execution of event k. The *global state* of a system containing n processes is thus the n-tuple of all the local states. If we want to observe the system the most logical step to take is to look which events just have been executed at each process. Therefore we define the *cut C* of a distributed computation as a collection of the events that are executed at all the processes until then. The set of the last executed events form a n-tuple called the *frontier of the cut*. The natural graphical interpretation in our time-space diagram is a partitioning along the time axis (Figure 3). The events that are executed before making the cut have occurred in some global time order. To be able to say something about this order we introduce the concept of a *run*. A run is a ordering of all the events in the global history, and is consistent with all the local histories. This means that the events belonging to a process P_i occur in the same order in the run as they occur in the process. Causal relations between events in different processes are however not taken in account, and a run therefore could correspond to an impossible execution sequence.



e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
P_1
P_2
P_3
e_2^1 e_2^2 e_2^3
e_3^1 e_3^2 e_3^3 e_3^4 C e_3^5 e_3^6 C'

Figure 3 Cuts of a distributed computation

Knowing all this we can try to obtain the global state by a monitor process P_0. An active monitor sends 'state inquiry' messages to each process. The processes send upon receipt of this message their local state σ_i. The monitor can thus construct the global state after it has received all the local states of the processes, that in fact define a cut. Another possibility is a passive monitor process: here all processes send a message to the monitor upon finishing an event. Both approaches may however lead to serious problems. The monitor process is part of the system, so there are delays on all the messages that are exchanged. This means that messages are not delivered at the same time at all processes and that the cut that is obtained may never have occurred during the examined run. Using this cut can thus lead to incorrect conclusions about the state of the system. It is essential that the global state that is obtained is consistent, in other words that it could have been constructed by an idealized, external observer who also respects the causal relations between event belonging to different processes.

5.3 Consistency

A cut is consistent if for all events *e* and *e'*: if *e* is part of the cut and *e'* causally precedes *e* implies that *e'* also is part of the cut. In other words, an event must not already been observed while it is causally precedes by another event that has not yet been observed itself.

$((e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C.)$

In the time space diagram this means that all arrows that intersect the cut must have their bases to the left and their heads to right of it, otherwise the cut is inconsistent. So in Figure

3 cut C is consistent while cut C' is inconsistent. A consistent global state is one corresponding to a consistent cut. So a consistent global state is a global state that could be constructed by an idealized observer external to the system. A run is called consistent if for all events, event e causally preceding event e' implies that event e occurs before event e' in the run. A consistent run can also be described as a sequence of consistent global states. Each global state Σ^i is obtained from the previous global state Σ^{i-1} by the execution of event e^i. Because there exist multiple processes next to each other, more than one event may be possible from some global state, resulting in multiple possible consistent runs. So from a global state it may be possible to execute more than one event, each leading to a different global state, where the process is repeated. This way all possible consistent runs can be graphically depicted in a lattice, consisting of global states that are interconnected if a event can lead from the one global state to the other.

Concluding, if we want to construct a global consistent state of the system, the sequence of events that are observed at the monitor process have to correspond to a consistent run. This also called a consistent observation of the system.

The reason that an observation of the global state of the system is not always consistent, is that due to delays messages may be delivered out of order. The order of event messages sent between two single processes can be easily restored. All the messages are tagged with a sequence number, and the messages can be delivered in the correct order at the monitor for example. This results in a First In First Out (FIFO) delivery rule. Such a FIFO delivery rule is sufficient to guarantee that observations correspond to runs. The causal precedence relations of events belonging to the same process are respected. The FIFO delivery rule is however still not sufficient to guarantee consistent observations. Events of different processes may still be delivered out of order at the monitor process.

Ideally we would like to have a global clock that could be used as a timestamp of the messages. When some event e causally precedes event e' this implies that the timestamp of event message e is smaller than e'. We will call this the *clock condition*. In our asynchronous system a global clock is of course not possible. There is however a mechanism to obtain the characteristics of a global clock in a asynchronous system.

Each process maintains a local variable LC called its *logical clock*. The value of the logical clock when event e_i occurs is denoted $LC(e_i)$. Each message m that is sent contains a timestamp $TS(m)$ which is the logical clock value that belongs to the sending event. When a receive event occurs, the logical clock is updated to be greater than both the previous local value and the timestamp of the local message. Otherwise the logical clock is simply incremented.

$$LC(e_i) := \begin{cases} LC + 1 \\ \max\{LC, TS(m)\} + 1 \end{cases}$$

In this way also the causal precedence relation is preserved, and a consistent observation of the asynchronous system can be achieved. This is done by having a lower level service layer deliver the received messages in increasing timestamp order to the process, because event e causally preceding e' implies $LC(e)<LC(e')$. The only problem is that a message may never be delivered in fear of receiving an other message with a smaller timestamp. The logical clock lacks gap-detection. The solution is to use FIFO communication on the single interconnection between two processes. Then if a process receives a message from process P_i with timestamp $TS(m)$ it is at least certain that no other message will be received from process P_i with a smaller timestamp. If from all other processes a message has been received with a timestamp greater than $TS(m)$, then the message is said to be stable, and may be delivered safely.

5.4 Vector clocks

Delivering all received messages that are stable in increasing timestamp order may introduce unnecessary delays. The delay is unnecessary if there still have to come messages with smaller timestamps but that these are related to concurrent events. So events that are causally unrelated may be forced to wait for each other. The timestamps used until now do not say enough about the causal precedence relation. If $LC(e) < LC(e')$ then event e may causally precede e' but they can also be concurrent. It is only known for certain that e' can not causally precede e. What we would like to have is another timing mechanism say TC so that event e causally preceding event e' is equivalent to $TC(e) < TC(e')$. We will refer to this as the *strong clock condition.*

Strong clock condition: $e \rightarrow e' \equiv TC(e) < TC(e')$

A straight forward way of achieving this strong clock condition is to produce for each event e a set of all events that causally precede it, and tagging it to the event as a 'clock' value. This is called the causal history $\varphi(e)$ and is in fact the smallest consistent cut that includes e.

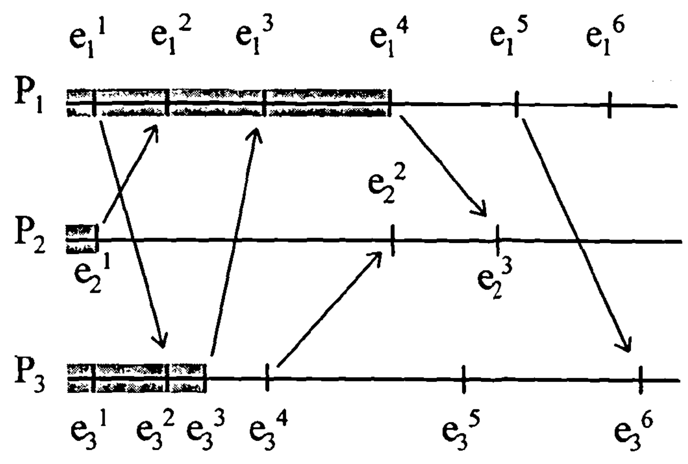


Figure 4 Causal history of event e_1^4

The maintenance of the causal history is simple. Each process begins with an empty causal history. On execution of an event this event is added to the causal history. If the event is the receipt of a message from another process, the causal history is the union of the causal history related to the message and the causal history of the previous local event. From the definition of causal histories it follows that: *e* causally preceding *e'* is equivalent to $\varphi(e)$ being included in $\varphi(e')$. Hence examining the strong clock condition can be done by checking if the event is an element of the causal history of the other. The only problem is that causal histories grow rather fast if they are entirely recorded. All the processes consist of a concatenation of events that can be numbered sequentially. The causal history can be represented in the following way: of each process the last event that belongs to the causal history has to be recorded. This results in a *n*-dimensional vector of events *VC(e)*, for *n* processes. The events can uniquely be identified by their sequence number at each process. So if the second item of a vector is four, then this means that the first four events of process P_2 belong to the causal history of the event to which the vector is related.

The resulting mechanism is known as vector clocks. Each process P_i maintains a local vector *VC* of natural numbers where $VC(e_i)$ denotes the vector clock value when event e_i is executed. This vector clock value is used as a timestamp when sending messages. From the foregoing we can derive the following updating rules for the vector clock of process P_i:

- If e_i is an internal or send event then the local component of the vector clock is updated: $VC(e_i)[i] := VC[i]+1$
- If e_i is a receive event then each component of the timestamp of the message and the local vector clock are compared, and the maximum is assigned to new vector clock. After that the local component is increased by one. Thus: $VC'(e_i) :=$ max.$\{VC, TS(m)\}$ and $VC(e_i) := VC'(e_i)+1$.

Now we can read some important facts from these vector clocks. We define that vector *VC* is less than *VC'* if and only if all elements of *VC* are smaller or equal to the same elements in *VC'*. Now causal precedence between events can be checked by the following rule:

$$e \rightarrow e' \equiv VC(e) < VC(e')$$

Note that this is the same as the causal history of event *e* must be contained in the causal history of event *e'*. If it is also known which two processes are involved, only the two components related to the sending process have to be compared:

$$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$

Meaning that at process P_j already a message has been received to which event e_i causally is related. Following the definition above two events e_i and e_j are concurrent if and only if:

$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

Even the consistency of a cut can be determined. Remember a cut is consistent if in the time space diagram there is no edge that begins outside the cut and ends inside the cut. The frontier of the cut consists of a vector with *n* components $(c_1,\ldots,c_n)$. Thus we have to check if the causal histories of the events in the frontier of the cut contain events that lie beyond the frontier of the cut. In terms of vector clocks, a cut is consistent if and only if:

$$\forall i,j\colon\ 1 \leq i \leq n,\ 1 \leq j \leq n\colon\ VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i]$$

The vector clock mechanism even has some weak form of gap detection. Suppose that the *k*-th component of the vector clock of event e_i is smaller then the same component of the vector clock of event e_j. This means that e_j has a message from process P_k in its causal history that has no causal precedence to event e_i. Then you know that there has to be a event e_k that causally precedes e_j but not causally precedes e_i. In formula:

$$\textit{if } VC(e_i)[k] < VC(e_j)[k] \textit{ for some } k \neq j \textit{ then } \exists e_k\colon \neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Thus now we have a clock mechanism that can be used to say something about the global state of the system, but without the use of a real-time global clock. In the following paragraph we will discuss how we can implement causal delivery at the processes using the vector clocks and how this can be used to get a view of the global state of the system.

5.5 Getting a view of the system: distributed snapshots

Now we finally can show that is possible to obtain a consistent global view of the system based upon local information. First we will give a solution for a passive monitor process constructing the global view. Each process updates its vector clock only for events that will be notified to the monitor and this vector clock is used as a timestamp *TS(m)* of the event message. At the monitor process it is important that the messages of all the processes are delivered in causal order, because in this way a consistent observation always will be the result. This can be done for a message *m* from process P_j by checking if:

1. there are no earlier undelivered messages from P_j
2. there is no undelivered message *m'* from the other processes such that the sending of this message causally precedes message *m*.

The first condition is satisfied when there are *TS(m)[j]-1* messages delivered from process P_j. For the check of the second condition the weak gap property of the foregoing paragraph can be used. Suppose that we take *k=i*, then it can be checked if there exists a message *m'* that causally precedes the examined message. This is done by checking if there is a component of the timestamp *TS(m')* that has the following property: *TS(m')[k] < TS(m)[k]* for some *k* not equal to *j*. If this is the case then there still exists an event with corresponding message *m'* that causally precedes the examined message. Thus no undelivered message exists if *TS(m')[k]≥TS(m)[k]* for all *k*. These tests can be implemented at the monitor process by maintaining an array where the *i*-th component stands for the number of the last message received from process P_i.

At this point we have a structure to obtain a consistent view of the system by letting the processes send messages for important events, and having them causally delivered at the passive monitor process. This can however be done at every process rather then just at the monitor process, and the mechanism above can be used to implement some form of causal broadcast.

Using causal delivery an active monitoring process can also be developed, assuming that the network is strongly connected. In this case the monitor process itself sends a 'take snapshot' message. When a process receives its first 'take snapshot' message it stops executing events on behalf of the computation. Furthermore it relays the message to all processes connected to it. After this the process waits until on all incoming channels 'take snapshot' messages have occurred, and records the other messages in between. Now the process knows that all the processes in its environment know about the take snapshot message and its contribution to the monitoring mechanism is completed. It now sends its local state and the recorded messages to the monitor process. The monitor process can reconstruct a consistent global state using the responses of all the processes.

5.6 Conclusions

It is obvious what the problems are in distributed computing systems. The only way to communicate to each other is by means of messages over the network. These messages can be out of order, and therefore the observation of the system is not always one that actually could have occurred. We have seen however that it is possible to reason at a local process about the global state of the system, only based upon information that is locally gathered. To answer our question from the beginning of this chapter: Yes, theoretically it is very well possible to construct a consistent global view of an asynchronous distributed computer system. Additionally we have gained some insight into the problems involved in concurrent processing, whilst preserving the happened before relationships between processes.

Suggested literature:

[mul93] - This chapter is based on chapter four of this book. The book discusses the subject of course in a more detailed and formal way. In chapter five the techniques are used to create a causal broadcast algorithm, which is not discussed in this thesis.

6. Remote Procedure Calling

Most interchange between processes in a distributed computing systems consists of remote operations. One process sends a request to perform a certain action to another process. The other process processes this request and sends the results back to the requesting process. This is of course not a new concept, invented in distributed computing. In a centralized system the same mechanism can be found. Here applications make system calls that are performed by the operating system. The application can be viewed as the process requesting remote operations. The operating system routine can be viewed as the process that processes the request.

In a centralized system this interprocess communication is done by means of procedure calls. Why not use the same mechanism in distributed systems? The mechanism of procedure calls is a familiar one in programming. It acts as an interface that hides the sending of requests and replies between two processes. It would be nice to also have this in distributed computer systems. Remote operations being performed through the same procedure call mechanism. This would hide the difference of locally made procedure calls, and calls made to remote processes. The applications making the call do not notice whether the call is processed locally or at a remote process. This is one of the reasons why Remote Procedure Calling (RPC) is used.

Another reason why RPC's are used, is to deal with heterogeneity in for instance processor types or used programming languages at different nodes in the distributed network. The request made by a process can be translated into the uniform language used by the RPC interface. At the side of the receiving process this uniform RPC language can be translated back into a form that is understood by that process. This is called marshaling.

6.1 RPC model

Now let us briefly look at how this RPC interface is modeled. The application or process that makes the request will be called the client. This client calls a subroutine that is executed in another application or process, called the server. It is now therefore obvious why RPC is also called client/server communication. The client does not call this subroutine directly. It calls another subroutine at the clients location called the client stub. This client stub translates the request to an uniform language used within the RPC mechanism. The resulting request is passed on by the RPC transport service over the network to the RPC transport service at the server. Here the request is translated into a request understood by the server. Results are sent back using the same mechanism. At the client the results are translated back by the client stub. For the client the whole thing looks like a regular procedure call. The model is depicted in Figure 5.

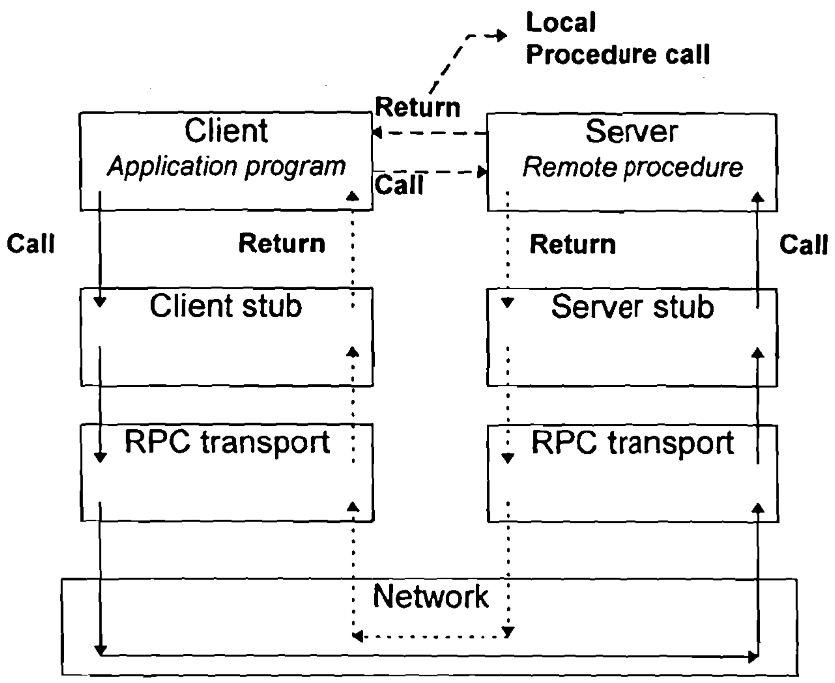


Figure 5 Remote procedure calling model

Although the RPC has the same structure and feel of a regular procedure call, there are differences. When during a local procedure call the server crashes, the client application that made the call automatically crashes to. No code is needed to deal with this situation. During a RPC both client and server may fail independently. In the RPC interface there must exist some exception handling to deal with this.

Another consequence of client and server being separated, is that they do not operate in the same memory address space. Values can be passed on with the procedure, but elements like pointers, global variables and functions are much more complicated. Pointers can for instance be marshaled by the RPC interface by passing on the data item that is pointed to. This is however not always the most convenient way. Suppose that the data item is a long array wherein the pointer points to a certain element of the array. Because it is not known at the client how the server operates, it is not known whether the server needs the entire array or just the element pointed to or the elements that follow after it.

A more convenient way is to let the server make a RPC back to the client when it runs into a pointer, to obtain the necessary data. This avoids transferring unneeded data across the network between clients and servers. Functions are passed to a procedure call by passing a pointer to that function. Passing functions to a RPC like this is almost impossible. The entire function could be copied to the server, but then the hardware of client and server should be the same, programming code of the function must be position independent etc. Yet another option is the use of so called stateless servers. These servers just perform the requested actions, but do not remember anything for the client. The complexity is transferred to the clients side that now is responsible for the parameter handling and, in case of more complex processes, the storage of intermediate results and states of the process.

As we have seen can RPC's provide us with a well known mechanism to perform remote operations like we have always performed local operations. Using the remote procedure call mechanism client applications do not see the difference between local and remote operations. Furthermore it is a tool that can be used to deal with heterogeneity in the system.

Suggested literature:

[ber87] - A discussion about how RPC's can specifically deal with heterogeneity in a distributed computer system.

[bir84] - This early article discusses how the authors have developed and implemented a RPC package. This article mainly presents the practical side of the story, and was one of the first (if not the first) article about RPC's.

[mul93] - Chapter 9 about interprocess communication also discusses the RPC mechanism.

[wil87] - An in-depth discussion about how to build distributed systems using remote procedure calls. The paper also discusses those implementation decisions which affect transparency and the used distributed applications.

7. Replication

Replication can be described as the maintenance of on-line copies of data or other resources. The motivations for replicating data and/or resources across the system are the following:

Performance enhancement: replication can be used to enhance service response times. Data that is shared between a large number of clients should not be held at a single server, because then this server would act as a bottleneck. It is preferable to distribute copies over the network so that it is close to all users.

Enhanced availability: if data is replicated over multiple failure-independent servers, then a client can access another server with the same information when its default server fails. Suppose that the chance of a single server failing is p. Now if we have say n replicated servers across the network then the chance of all servers failing, and thus total unavailability, is p^n which is much smaller.

Fault tolerance: if a collection of servers processes a request of a client, then some fault tolerance can be guaranteed even if one or more servers would fail. Suppose that a failing processor just would stop processing requests. Then in order to make a τ-fault tolerable system you need at least $\tau+1$ replicated servers. If a client makes a request to a failing server, and it does not receive an answer, there are still τ other servers that can service the request. In case of servers making arbitrary errors, for example sending erroneous messages, $2\tau+1$ servers are enough. A client sends a request to all servers, and accepts the result of the majority of the servers. This because when τ servers fail, $\tau+1$ servers still deliver the correct result.

The key problem with replicated data is again consistency. If several clients make requests that involve updates and reads of the same (replicated) data it is normally not acceptable that different clients obtain different values of the same data. At least not in the cases where that might lead to inconsistencies between different applications. It is not only important that updates are made at all copies, but also the order in which these updates happen are of importance. Like in the case of obtaining a consistent global view of the system (see chapter 5) causal precedence must be preserved. In certain circumstances some relaxations can be made. For instance order is not of interest in the case of read-only data, or concurrent requests.

7.1 Basic architectures

There are some basic forms of replication management. Here we will define a model in which these different architectures can be described. Furthermore we will look at some basic replication management structures. In the model the replicated data is held by distinct replication managers (RM). It could be modeled such that clients make requests to replication managers directly. We however introduce a Front End (FE) between the client and the replication manager that is responsible for talking to the replication managers. This is a way in which location transparency of the data can be implemented, because the client does not have to address the data explicitly. The basic model is shown in Figure 6.

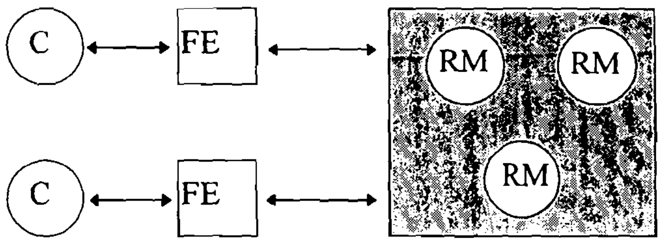


Figure 6 Basic model for replication management

In the following paragraphs we will shortly discuss two basic architectures for replication management. There exist many in between forms of these architectures, but the two given show the possibilities for replication well enough. These two types of architectures are called *'primary backup'* and *'active replication'* respectively. Along with active replication, another replication scheme called *gossip replication* scheme will be discussed.

7.2 The primary backup architecture

In the primary backup architecture there are again several failure independent replication managers. The basic idea of the primary backup architecture is that one replication manager is designated as the *primary* and the others as *slaves*. The clients make requests only to the primary. If this primary fails then one of the backups takes over, and becomes the new primary. We will call this *failover*. Updates of data are made at the primary only. The primary then relays the updates to the slaves (Figure 7).

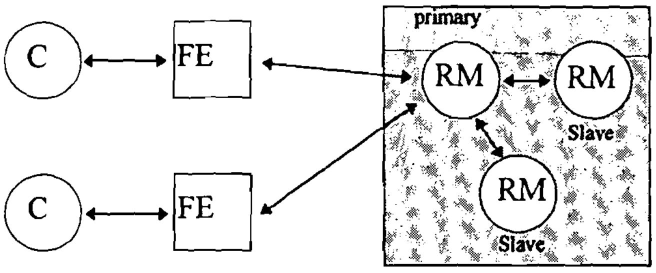


Figure 7 The primary backup architecture

So there are several properties that must be satisfied by a primary backup protocol. The first one is rather obvious.

Pb1: At any time there is at most one server that is the *primary.*

Furthermore each front end has to remember which server is the primary so:

Pb2: Each frontend maintains a server identity of the server to which it makes its requests.

Of course it possible that the server that is thought to be the primary is no longer the primary any more. So the request of the frontend reaches a broken down server or a slave, and hence the request should not be processed.

Pb3: If a request arrives at a server that is not currently primary, then that request will not be processed.

Until now we have specified a protocol for the frontends interactions with the service, but have not said anything about the obligations of the service to perform its task. The above three properties could for instance be implemented by having a server that just ignores all requests. We assume that after every request a response should follow. If a frontend has sent a request and no response follows, then we speak of an *outage*. Now the last property that expresses the obligations of the replication manager is:

Pb4: In a given period of time there may only occur a fixed amount of outages.

The four properties above more or less define the primary backup protocol. The only problem is that no communication link is perfect and that a replication manager can fail. So there have to be made some precautions. The loss of messages across links can be detected by not having received a response after some time-out. The crash of the primary is a bit more complex, because it is not known if it was processing a request and if it has already sent updates to all slaves. The updates are always sent to all slaves, so the new primary can inform at the other slaves what the last message was they received. If it finds out from the others it missed an update it performs the update himself, and also send it back to all slaves. In this way everything is back to normal and the latest client that still had no response of course can be served.

7.2.1 Conclusion

The advantage of the primary backup protocol is that it is rather simple. All frontends talk to one single replication manager, the primary. This primary decides on his own which updates are made and sends these to the slaves. In this way all the slaves do not have to agree on some update. Only some effort has to be put into the reliable transmission of these messages and a correct failover protocol.

There are some drawbacks of the primary backup protocol. The primary still is located in one place and in this way data still can not be distributed close to all clients. All clients still have to access this single, possibly remote, replication manager. Furthermore requests can be lost by the failure of the primary. When the primary has just received a request and then fails the client never gets a response, and hence the client should maintain a timer and try again after some time.

7.3 The active replication and gossip architecture

As we have seen, the two biggest drawbacks of the primary backup architecture are that: a request can get totally lost at a failing primary replication manager, and there is always a single replication manager that is accessed by all frontends. What we would like is an architecture where more than one replication manager can be accessed by a frontend. This means that the frontend can make the request to the closest replication manager. Furthermore it could make the same request to multiple replication managers. In this way making a request to $2\tau+1$ replication managers and choosing the majority a τ fault tolerable service can be created. We will discuss two architectures that look much alike; the active replication architecture and the gossip architecture.

In the active replication architecture a request is made to all the replication managers. All replication managers update their state accordingly and send their response back. The active replication architecture is shown in Figure 8.

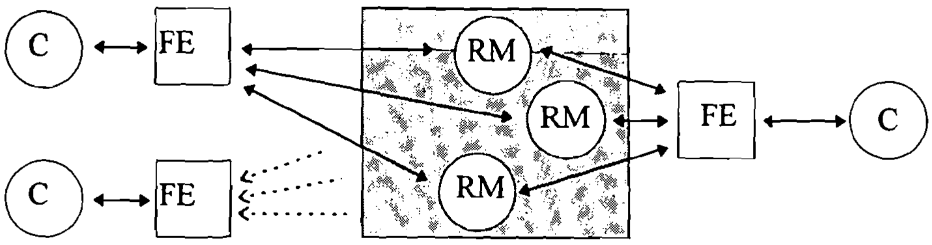


Figure 8 The active replication architecture

In the gossip architecture the frontends make a request to the closest replication manager. The updates ('the news') made to a replication manager are exchanged by that replication manager to the others by means of so called 'gossip messages'. It could however also make the request to more than one replication manager. The gossip architecture is shown in Figure 9.

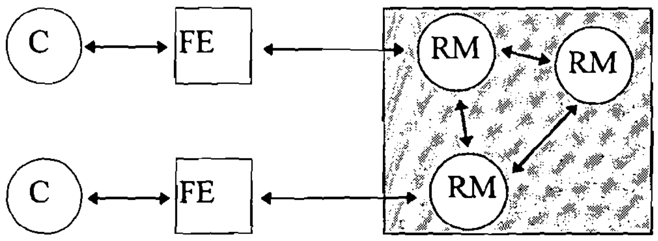


Figure 9 The gossip architecture

We can see that the two architectures look very much alike. The only difference is the place where the requests are exchanged to all the other replication managers. In the active replication architecture the clients frontend makes sure that all the replication managers receive the request. Drawback is thus that all replica managers must be accessed including the most remote ones, which is not all that good in terms of performance. In the gossip architecture the task of informing the other replica managers is performed by the accessed replication manager.

7.4 Replication coordination

Here of course the coordination between all the replication managers is the biggest problem. Every non-faulty replication manager should receive every update request. Furthermore the order in which the requests are processed can be of importance. The causal order between requests must be preserved. Every non-faulty replication manager receiving every update request can be implemented by using reliable broadcast protocols or agreement protocols. We will not go into the subject here, but it is discussed in chapter 5 of [mul93]. Here we will assume that a response should follow after every request, and hence lost messages can be detected.

Now some ways will be discussed in which replication managers are updated whilst preserving order. One of these leads to the gossip architecture discussed above.

Logical Clocks

In the case that causal precedence must be preserved, the logical clocks presented in chapter 5 can be used. These logical clocks or vector clocks are assigned to the messages as timestamps, and with these timestamps a message can be checked if a not yet received message could causally precede it.

Replica generated identifiers

Logical clocks are assigned by the requesting frontend. The biggest disadvantage of the use of logical clocks is that all processes (frontends and replication managers) must communicate with each other in order to be able to detect if a message is stable. In the case of replication generated identifiers the replication manager assigns an unique identifier (what's in a name?). Here the only communication required is between the requesting frontend and the replication managers. Upon receipt of a request a possible candidate unique identifier is sent to the other replication managers. After this an unique identifier will be chosen by the replication managers. One simple solution for a system of processors that halt in case of failure is the following:

- Every *i*-th replication manager maintains two variables
    - $SEEN_i$: the largest <u>candidate</u> identifier assigned to any request so far seen by the replication manager.
    - $ACCEPT_i$: the largest unique identifier assigned to any request so far accepted by the replication manager.

- Upon receipt of a request each replica manager computes the candidate unique identifier as: max. $(SEEN_i, ACCEPT_i)+1+i$ (This means that all candidate unique identifiers themselves are also unique).

- After the candidate unique identifiers of the other replication managers are received by using some agreement protocol, the unique identifier is chosen to be the maximum candidate unique identifier.

Now we still have to check when a request with an accepted unique identifier is stable. If we have used the mechanism above to assign unique identifiers, a request is stable iff: there exists no other unaccepted request seen by the replica manager, with a candidate unique identifier that is smaller or equal to the unique identifier of the examined request. If there were such a request then the unique identifier that will be accepted for it could end up smaller, and thus the request should be executed earlier. If there is no such request then the mechanism above makes sure that requests that are seen later are eventually always assigned a larger unique identifier. After requests are stable, they can be executed in an ascending unique identifier order.

Although not all frontends have to communicate, these replication generated identifiers cost a lot of communication between the replication managers. Also some agreement or reliable broadcast protocol is needed.

Gossip messages
Here we will examine the way in which replica managers are updated in the gossip architecture. The frontend normally communicates with a single replication manager. The replication managers update one another by exchanging gossip messages which contain the most recent updates they have received. These gossip messages can be exchanged only occasionally, after several updates have been collected. They can also be exchanged when a replication manager finds out that it is missing an update which it needs to process a request, but that has been sent to one of its peers. Replication managers are able to find out that they are missing an update because client front ends keep a sort of version number of the latest accessed data, which is sent along with the request. Control may be given back to the client immediately after the request or, for increased reliability, or clients may be forced to wait until the update has been delivered to $k+1$ replication managers and thus will be delivered everywhere despite up to k failures.

In order to control the ordering, each frontend keeps a vector timestamp *prev.* which reflects the version of the latest values accessed by the frontend and is sent with every request. When a request is returned a new timestamp is included by the manager. The *prev.* timestamp is then updated with the new timestamp.

So in order to be able to exchange gossip messages the replication managers contain the following components:

- *value*: this can be seen as the current state or contents of the replication manager. Every replication manager starts in a (possibly empty) state. After that its contents are changed by update requests from frontends and other replication managers and hence its value changes accordingly.

- *value timestamp*: this is the timestamp that represents the updates that are reflected in the value. It is also updated whenever an update is made to the value.

- *update log*: All updates are recorded in this log as soon as they are received. This is because updates may not be executed because the update request is not yet stable. Furthermore updates are kept for security reasons until the replication manager knows that the update has been received by all other replication managers.

The value timestamp of the replication managers are constructed in the following way. If there exist n replication managers then the value timestamp is a n-dimensional vector. The ith element of replication manager i corresponds to the number of updates received from frontends by i; and the jth component equals the number of updates received from frontends by replication manager *j* and sent to *i* in gossip messages. So a value timestamp (1,2,3) at manager 1 means that the first update has been accepted from a frontend at manager 1. Furthermore the first two updates from frontends at manager 2 are also accepted at manager 1 because they are propagated to it in gossip messages. The same goes for the first three updates performed at manager 3.

Let us first look at how a query request is handled. Every request contains the *prev.* timestamp along with the operation that has to be performed. This timestamp shows how recent the values collected at the frontend are; and thus the replication manager should return a value at least as recent as this one. So in terms of timestamps: the *prev* timestamp of the query must be smaller then or equal to the *value* timestamp of the replication manager. The manager keeps the query on a pending list until this condition is fulfilled, either by update gossip messages or by a request for information to the other replication managers. For example, if the *value* timestamp at the replica manager is (2,3,6) and the *prev* timestamp of the query is (2,2,8) then it can be seen that two updates from replica manager 3 are missing. After the request has been responded to, the frontend updates its timestamp according to the new timestamp.

When replication manager *i* receives a update request it increments element *i* in its replica timestamp by one. Then the request, containing the *prev* timestamp and the update operation, should be placed in the log and assigned an unique identifier. The request in the log is given an unique identifying timestamp *TS* by replacing the *i*th element of the *prev* timestamp of the request by the *i*th element of the replication timestamp. The *TS* timestamp is immediately sent back to the frontend where it is merged with the old *prev* value. If the frontend has sent the request to more than one manager, it should do so for every *TS* that is sent back. Within the replication manager the update can be executed as soon as the updates upon which it depends already have been executed. This again means that the *prev* timestamp of the query must be smaller than, or equal to the *value* timestamp of the replication manager, like in the case of a query request. Using this mechanism update operations are performed in causal order. Note that this mechanism is derived from the vector clocks used in chapter 5.

7.4.1 conclusions about replica coordination

As we have seen above there are several ways in which coordination among replicas can be constructed. With the use of logical clocks all messages exchanged are processed in causal order at all the replication managers. The big disadvantage here is that all processes including all frontends have to communicate to each other. Even frontends that have no work must exchange messages in order to make sure that requests at replication managers become stable after some time so they can be executed.

In the case of replica generated identifiers the frontends that have not made the request do not have to participate in the process. The replication managers however have to reach some agreement about the identifier to be used, using some agreement protocol. This need for agreement involves a lot of expensive communication. A big advantage could be that request has the same identifier over the entire network, so total order is implemented.

The gossip architecture is some lazy form of active replication. Requests are held on at a replication manager until it knows for sure that the requested value is recent enough and then returns it. Other replication managers are updated in some lazy fashion using gossip messages. This architecture uses a lot less communication but the price is paid in the degree of performance. Because of the lazy fashion in which updates are made requests can take much more time. Furthermore the danger exists of making updates based upon (too) old queries.

7.5 Conclusions

As we have seen there are different forms of replication management. The primary backup method is the simplest. It involves relatively little communication for updating the slave replication managers. Only some effort has to be put into the takeover by a slave after the primary has failed. The major drawbacks are that only a single replication server can be accessed and that requests can get lost by a failing primary.

The other architectures where every replication manager can be accessed are much more complex. The reason for this is that every request has to arrive at every replication in the correct order and this while every manager can be accessed. Hence all these architectures are rather expensive in terms of communication. It is however in some cases possible to relax some of the demands, and gain something in performance. Furthermore the amount of communication, availability, fault tolerance and performance can be exchanged by using some other subform of replication management better suited for the specific task..

Suggested literature:

[mul93] - In chapter 7 the active replication mechanism is discussed, while in chapter 8 the primary backup approach is discussed. These chapters have been the foundation for the presentation of these schemes in this thesis.

[cou94] - Chapter 11 of this book discusses replication management. The basic architectural models presented there are used here. In this chapter also the gossip architecture is presented. Furthermore it elaborates upon how updates can be performed in causal order at the replication managers.

8. Transaction management

In the last chapter we discussed how data can be distributed in a computer system. We have not looked into how clients use the available data. In a distributed computer system more than one client may have access to the same data. So operations in a data server performed on behalf of different clients may sometimes interfere. A sequence of operations that form a single step, transforming the server data from one consistent state to another is called a *transaction*. A simple example of a transaction is a banking transaction. The data of an account can be accessed by all bankoffices, to either withdraw or deposit money to the account. A transaction normally consists of write and read operations. If two or more transactions that involve the same data component take place at the same time, then it is very well possible that the one transaction reads a wrong value because another transaction just adapted the old value but did not yet write the new correct value.

Thus it is important the client sees a server that is free from interference from concurrent operations being performed by other clients, and one that either completes an entire operation successfully or does not complete it at all. In other words it needs to make its operations *atomic* in order to keep its data consistent. The results produced by this server must be the same as the results of a server that performs transactions sequentially. In this chapter we will first discuss how transactions can be managed that involve a single data server. Important here is how concurrency between different clients is handled and how the occurrence of deadlocks can be dealt with. In the next chapter we will discuss how distributed transactions can be managed and the associated distributed deadlocks can be solved. In the last part of that chapter we will shortly discuss how transactions can be managed in an environment with several replicated data servers.

8.1 Transactions

As we have said a transaction is a sequence of operations that form a single step, transforming the server data from one consistent state to another. Such a transaction may consist of several operations/messages exchanged between a server and a client. This interaction can be seen as a conversation. Each conversation between a client and a server gets an identification number in order to be able to distinguish between different conversations.

As we have seen transactions should have some properties like for instance atomicity. These properties are often combined in the so called *'ACID'* properties.

This stands for:

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity means that each transaction appears to occur either completely or not at all ánd that partial effects can not be seen by other transactions. So the first aspect of atomicity involves the all-or-nothing behavior of a transaction. This means that the effects of a transaction are atomic even if the server fails, which is called *failure atomicity*. Furthermore this also implies that if a transaction has completed successfully that the results are stored in some permanent storage and are very likely to survive later failures. This is the *durability* property from above. To support both failure atomicity and durability it is needed that the data is recoverable e.g. that in case of a hardware or software failure all data from completed transactions is available on some permanent stable storage. The second aspect of atomicity is *isolation* meaning that other concurrent transactions do not see any intermediate effects of the executed transaction. With the *consistency* property here is meant that the application that performs a single transaction takes the data from one consistent state to another. It is however the responsibility of the users of the database that their different transactions maintain the consistency of the database.

An atomic transaction is achieved by cooperation between a client program and the server. A client will request the transaction as a sequence of operations added with an *'open transmission'* operation and a *'close transmission'* operation. On receipt of an *'open transmission'* operation, the server assigns a identifier to the transaction and returns that to the client. A client can thus use this identifier to tag the operations in the rest of the transaction. Eventually there are two possibilities: the server completes successfully and commits, or the server fails in some way and aborts the transaction. When a server has committed or aborted then this can be reported to the client. The client can also abort by sending an abort transaction message.

If a server fails then all transactions that were not yet committed are aborted by the server when it starts up again. The client will become aware of a server failure after an operation returns an error message after a time-out. If a server fails and starts up within this time the transaction and hence the transaction identification will not be valid anymore, and the client will be informed. The failure of a client can be detected by assigning time outs to each transaction.

The aim for any server that is involved in transactions is to maximize concurrency between different transactions. This can however lead to situations where different transactions interfere with each other. We will see two examples of how things can go wrong, the first is known as the lost update problem and the second as the inconsistent retrieval problem.

The lost update problem:
Suppose that Transaction X wants to transfer $4 from account A to account B and that transaction Y wants to transfer $3 from account C to account B. Then the following situation could arise.

Transaction X	account A	account B	account C	Transaction Y
Read A = 100	100	200	300	
Write(A-4)=96	96	200	300	
	96	200	300	Read C = 300
	96	200	297	Write(C-3) = 297
Read B = 200	96	200	297	
	96	200	297	Read B = 200
	96	203	297	Write(B+3) = 203
Write(B+4) = 204	96	204	297	

The owner of account B will not be pleased as his balance results in $204 in stead of $207. The update performed by transaction X is overwritten by transaction Y because it has never seen it.

The inconsistent retrieval problem:
Suppose that transaction X transfers an amount of money from account A to account B. Transaction Y wants to determine the sum of all accounts of the bank. The following situation could occur:

Transaction X	account A	account B	Transaction Y
Read A = 200	200	200	
Write(A-100)=100	100	200	
	100	200	Read A = 100
	100	200	Read B = 200
	100	200	sum(A+B) = 300
Read B = 200	100	200	
Write (B+100)=300	100	300	

The sum is calculated as $300 instead of $400 because transaction Y checks the accounts while the amount is withdrawn from one account and not yet deposited on the other. The retrieval of information is not consistent.

So although the transactions above are individually correct the results are not correct because of interference. What we want is an interleaving of operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time. This is then called a serially equivalent interleaving. In the following chapters we will discuss some methods for concurrency control while maintaining the transactions serially equivalent: locking, optimistic concurrency control and timestamps.

Even if serial equivalence is preserved, things can go wrong simply because servers and clients are allowed to abort transactions. This is for instance the case when transaction X writes a new value which is read by transaction Y. If transaction X aborts then this value that transaction Y uses is not valid anymore and transaction Y is not allowed to commit. So if there is danger for such a 'dirty read' then transaction Y should wait to commit until transaction X has committed or else abort. It can of course happen that if a transaction aborts another transaction has to abort, and another that is waiting for the second etc. This phenomenon is called cascading aborts. This can be avoided by only allowing reading of a data item if they are written by committed transactions. Also when a value is already written and the transaction aborts, the premature write must be replaced by the old value. Hence the server has to keep old versions of the data items to be able to correct premature writes.

8.2 Methods of concurrency control

As we have seen above, different concurrent transactions can interfere with each other. What we would like are methods to keep transactions serially equivalent. This means that the execution of the possibly concurrent transactions yield the same results as if they were executed in some serial order. If the order in which they are serialized is important, then it must be imposed upon it by the user. In the following paragraphs we will look into three methods of concurrency control: locking, optimistic concurrency control and timestamping. Concurrency control protocols are used to cope with conflicts between operations. Operations are said to conflict when their combined effect depends on the order is which they are executed.

8.2.1 Locks

A straightforward method to serialize operations of transactions is to use exclusive locks. If an operation in a transaction wants to use a data item, then the server tries to put a lock on it on behalf of that transaction. If the data item is already locked by another transaction then the request is suspended until the other transaction is ready and releases the lock. The transactions accessing a particular data item thus seem to be serialized with respect to each other.

Serialization however also means that all pairs of conflicting operations should be executed by two transactions in the same order. In the approach above this is true only if a transaction performs a single operation on the same data item, but not necessarily when more operations on the same data item need to be performed by the same transaction. Suppose that a transaction needs a data item, locks it, uses it, and releases it. Now based upon the previous use of the data item, it needs to be updated by the transaction, and should thus be locked again. Meanwhile another transaction could have locked the data item and the two transactions need not be serially equivalent anymore. This is solved by not allowing a transaction to gain a new lock after it has released one of its locks. This is called two-phase locking; there is a growing phase where locks are acquired and a shrinking phase where locks are released. Furthermore the problem of aborting transactions has not yet been addressed. In situations where transactions may abort, locks must be held until the transaction commits or aborts. This is called strict two phase locking.

There are some basic conflict rules: two read operations can not conflict because the data item is not changed and the result does not depend on the order in which they are performed. A read and a write just like two write operations might conflict, because the order in which they are performed might effect the results. Hence simply using exclusive locks for every operation reduces concurrency far to much. The operation conflict rules do tell us that:

1. If a transaction X has already performed a Read operation on a data item, then another transaction must not write this data item until X commits or aborts.

2. If a transaction X has already performed a Write operation on a data item, then another transaction must not read or write this data item until X commits or aborts.

So some locks are compatible to each other and some are not. In Table 1 the lock compatibility for a single data item is given.

	Lock requested	
Lock already set	*Read*	*Write*
none	Granted	Granted
Read	Granted	Wait
Write	Wait	Wait

Table 1 Lock compatibility

By using locks complying to the conflict rules above, inconsistent retrievals are prevented. Whenever a transaction performing an update tries to get a write lock while a transaction performing a retrieval already has a read lock on the same data item, then the transaction that wants to write the data item is delayed until the other transaction has completed and released its read lock. When the transaction performing an update has come first then the other transaction has to wait. In order to prevent lost updates something extra needs to be done. Recall that a lost update occurs when a read value is used to calculate a new one by two or more transactions because only the last written update will be preserved. Lost updates can be prevented by making later transactions delay their reads until the earlier ones have completed. This can be done by letting the transaction that first reads a value and then wants to use it to write a new one promote its read lock into a write lock. In this way later transactions are delayed until the new value has been written. If more than one transaction possesses a read lock on the data item then this strategy will not work because this could conflict with the fellow lock holders. This transaction has to request the write lock, and wait until the other read locks are released.

8.2.1.1 Deadlock

In a system where locking is used, deadlock can occur. As an example the following simple situation. There are two transitions that want to deposit money to an account. Both have read the balance and thus share a read lock on this account. Both want to write their new balance, but each has to wait for the other to release the read lock in order to promote it to a write lock. When no precautions are taken the two transactions will wait until there is a power failure. Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. The occurrence of deadlock can be made visible by a *wait-for graph.* In this graph every node stands for a transaction and an edge between transaction X and Y means that X waits for Y to release a lock. If a cycle occurs in this wait-for graph, then we are in a deadlock situation.

If one of the transactions in the cycle is aborted, its locks will be released and the cycle is broken. Hence deadlock is resolved. One solution to prevent deadlock is to lock all data items that are needed by the transaction when it starts. This will prevent deadlock because it will never have to wait for another transaction to release locks, but it clearly is not the most elegant way, and not always possible because all needed data items must be known in front. In this way data items may be unnecessarily long locked, and this decreases the effectiveness of sharing resources.

Another way to overcome the problem is to try to detect deadlocks. This can for instance be done by a lockmanager that locks and unlocks data items on request. It could maintain a representation of a wait-for graph, and detect cycles in it. This could be done at a certain time interval to reduce the overhead for the lock manager. If deadlock is detected, one of the transactions causing the deadlock is forced to abort.

The final and most commonly used solution is using time-outs. In this case a lock after some specified time can be broken by other transactions waiting for it. The transaction owning the lock until then usually must be aborted. The biggest problem is usually to determine the right time-out length. It is very well possible that there is absolutely no deadlock, but that a transaction took some more time. In this case transactions will be constantly aborted without a reason. Furthermore it might be possible that a transaction does not rely on the correctness of the locked data item. In this case a transaction is aborted while it could have committed. A solution for this problem is to ask the client to unlock the data item voluntarily, so it can commit when it chooses to.

It does depend on the situation which deadlock resolution is used. Time-outs are simple in use, but are not as simple to determine. Deadlock detection by a lock manager is more complex, but could have better results and be more fair to all transactions. In general using deadlock detection has the disadvantage that it induces an overhead on each transaction even when it is not always necessary. Furthermore locking can result in deadlock when nothing is done about it.

8.2.1.2 Deadlock avoidance using timestamps

Deadlocks can also be avoided using a combination of locking and timestamps. As soon as a transaction starts it is assigned a unique timestamp. The server numbers all transactions sequentially in the order they arrive. Now in case of a conflict between operations of different transactions, we determine if the later transaction is allowed to wait for the other transaction to release the lock or that it should abort based upon their timestamps. To do this we extend the lock compatibility with the following rule:

A transaction is allowed to wait for the release of a lock iff the transaction holding the lock has a smaller timestamp then its own, else it must be aborted.

Deadlock is avoided because transactions only wait for older transactions. Suppose for instance that deadlock could occur: then there must exist a cycle in the wait-for graph. Following the edges, the timestamps of the lockholders must be smaller after every edge. Eventually at the end of the cycle this would mean that the timestamp of a transaction must be smaller than its own, which is impossible. If multiple transactions are waiting for a lock to be released, they should be granted the ownership of the lock in timestamp order. This because else a transaction that was allowed to wait for the release of the lock must abort afterall when a younger transaction gets the lock first. Long transactions have a disadvantage using this scheme, because during their life they have a higher chance of conflicting with an younger transaction.

8.2.2 Optimistic concurrency control

In many situations two transactions accessing the same data item at the very same time are not very likely. So in stead of using the pessimistic approach by trying to prevent conflicts with every operation (for instance locking), one could use a more optimistic approach by letting transactions perform their operations like no conflict could occur. In this optimistic approach, when a client finishes its transaction with a close transaction request then it is checked whether or not a conflict has risen. In case of a conflict the corresponding transaction is usually aborted, and must be restarted by the client. Hence each transactions has the following phases:

Read phase:
In this phase every transaction keeps a backup version of all the data items it updates, that is not visible to other transactions. When a read operation is performed the backup of the data item is read if it already exists. If there is not yet a backup version, then the operation reads the (committed) data item itself. Write operations on data items are kept as backup versions.

Validation phase:
This phase starts after the client has requested the transaction to close. It is validated if operations performed on data items do not conflict with operations of other transactions. If the validation fails and there has occurred a conflict, the current transaction or the one it conflicts with needs to be aborted. Some conflict resolution has to take place here.

Write phase:
If the validation is successful then the new values given to data items by write operations have to be made permanent. If a transaction had only read operations then the transaction can commit immediately, else the transaction first has to write the new values of data items to permanent storage.

Each transaction that enters the validation phase is assigned a transaction number T_j in ascending sequence. This implies that a transaction always has finished its read phase after other transactions with lower numbers. This will help because during validation we want the transactions to be serially equivalent. The validation test is based on conflicts between operations of two overlapping transactions. In order to have two serially equivalent transactions T_i and T_j $(i<j)$ they must be conform to the conflict rules given in Table 2.

T_i	T_j	Rule
Read	*Write*	1. T_i may not read data items written by T_j
Write	*Read*	2. T_j may not read data items written by T_i
Write	*Write*	3. T_i may not write data items written by T_j and T_j may not write data items written by T_i

Table 2 Conflict rules for validation

Validation must test if rules 1 and 2 are obeyed. Rule 3 can be satisfied by not allowing more than one transaction in the write phase at one time. There are two ways of validating the transactions, forward and backward. In backward validation all the read operations performed by earlier transactions (lower transaction number) can not be influenced by writes of the transaction under validation, so rule 1 is satisfied. We thus serialize transactions in timestamp order. The validation checks if one of the read operations of the later transaction overlaps with one of the write operations of earlier transactions. If there is a overlap then the validation fails, because else the transaction would have read an old value that will be overwritten by another transaction that has been serialized as happened before the transaction under validation. The earlier transactions are from the transaction with the smallest transaction number that had not committed when the validated transaction entered the read phase, until the transaction with the largest transaction number when the validated transaction entered the validation phase. If for example transaction T_4 in Figure 10 is under validation, then it is compared with transactions T_2 and T_3. So the write sets of all earlier transactions that have committed but still might overlap with non validated transactions, have to be kept. The write set of T_l also has to be kept since there still is an active overlapping transaction.

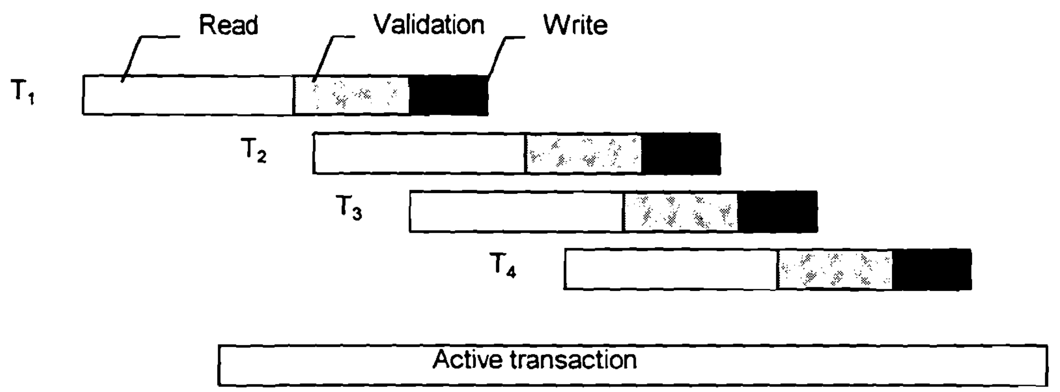


Figure 10 Validation of transactions

Forward validation checks the transaction undergoing validation with other later transactions that are still active and are still in their read phase. Rule 2 is here already fulfilled because the later transactions will not write until the current transaction is validated. This validation has to last as long as there are active transactions in their read phase. In Figure 10 the validation of T_l should thus last as long as the read phase of the active transaction. Hence active transactions have to be assigned identifications is order to be able to recognize when all overlapping transactions ended their reading phase.

The forward approach has some advantages: when a conflict is detected some sort of conflict resolution is still possible because one transaction is still active, in backward validation there is only the choice of aborting a transaction. In backward validation a possibly large amount write sets has to be stored and compared with one read set whereas in forward validation one write set must be stored and compared with active reads. Because read operation are often more common, read sets are larger compared to write sets. One drawback is that during forward validation new transactions may become active so that validation costs more time.

8.2.3 Timestamps

When using timestamps as concurrency control, each operation of a transaction is validated instead of the entire transaction. Using timestamps transactions can be totally ordered, by assigning a timestamp to a transaction when it begins. A transaction may only write a data item that is written or read only by transactions with smaller timestamps, and it may only read that data item when it is last written by a transaction with a smaller timestamp. Here with one server timestamps can be assigned using a clock value of the server or by assigning a timestamp each time a transaction is started.

Write operations to a data item by transactions are again kept in backup versions. These backup versions are invisible to other transactions until the transaction commits. The backup versions of a data item are not kept within the transaction but the server manages all the backup versions of a data item. So the committed version of the data item has a write timestamp attached to it, belonging to the last committed transaction writing it. Every backup version also has a write timestamp attached to it that is the same as the timestamp of the transaction writing it. Furthermore it has a read timestamp attached to it that is equal to the maximum timestamp of a transaction reading it. Whenever a transaction commits, the backup versions belonging to that transaction are used to overwrite the original committed version of the data items to become the new committed data items. The data item also gets the write timestamp of the backup version.

Now with this timestamp ordering a read or write operation of a transaction is checked whether or not it complies to the conflict rules. These conflict rules are given in table 3. Here a transaction T_j wants to perform an operation.

T_j	Rule
Write	1. T_j is not allowed to write a data item that already has been read by a later transaction T_i. Hence the timestamp of T_j must be greater than the maximum read timestamp of the data item.
Write	2. T_j is not allowed to write a data item that already has been written by a later transaction T_i. Hence the timestamp of T_j must be greater than the maximum write timestamp of the data item.
Read	3. T_j is not allowed to read a data item that already has been written by a later transaction T_i. Hence the timestamp of T_j must be greater than the write timestamp of the committed version of the data item

Table 3 Conflict rules with timestamp ordering

Whenever a transaction T_j wants to perform a write operation, the first two rules are combined in the following test. First it is checked if the timestamp of T_j in greater than the largest read timestamp of the data item, secondly it is checked if the timestamp of T_j is greater than the write timestamp of the committed data item. If these checks are successful then the write operation is performed by writing to a backup version, else the write is too late and should be aborted. If a backup version belonging to T_j already exists then this one is overwritten, else a new one is created.

In case of a read operation the following procedure is followed. If the timestamp of T_j is smaller than the write timestamp of the committed version, than a later transaction has already overwritten the data item and the read is too late. The transaction should be aborted. If the timestamp of T_j is bigger than the write timestamp of the committed version, then the data item still may not be immediately read. This is because there may be other transactions that have not yet committed but have started earlier, that perform write operations on that data item. Thus in order to keep the data item consistent, the backup versions of a data item should be committed in timestamp order. This means that read operations have to wait until there are no more write operations possible on that data item by earlier transactions. Hence the backup version of the data item, if any, with the largest write timestamp smaller than or equal to the timestamp of T_j is selected. When this selected version commits, or is the committed data item, than the read operation is performed on the last committed version. Else if the transaction belonging to the backup version aborts, the read operation is performed again. Note that if the same transaction has already written a backup version of the data item, this one will be used. It is also not possible that another write operation with a timestamp larger than the selected one but smaller than the read timestamp will occur, if the largest upto then already has been selected. This because with a write operation it is checked if the timestamp of this operation is greater than the maximum read operation.

In this way always a committed version is read. So there is no danger of reading a data item written by a transaction that is going to abort, making the read data item useless or faulty. Furthermore deadlock can not occur because the transactions are always waiting for an earlier one. In situations where there are mostly reads, the timestamp ordering method is beneficial. This is because read operations are less restricted. They only have to belong to a transaction that is started later than the transaction that has written the latest committed version and wait until earlier writes have committed. Write operations however must occur later than the latest read operation and later than the write timestamp of the committed version.

8.3 Conclusion about methods of concurrency control

Timestamp ordering and two phase locking are both pessimistic. They detect conflicts between transactions every time a data item is accessed. The ordering with timestamp ordering is transaction based, every time a transaction starts it is assigned a unique number. The ordering of two phase locking is based upon the access of a data item. Whenever a conflict is detected, timestamp ordering will abort the transaction immediately whereas locking waits, but possibly still aborts the transaction later. Timestamp ordering is better in situations with more read than write operations. In locking schemes the write lock is much stronger. So in situations with many updates compared to reads, two phase locking is preferable. Optimistic concurrency control is very efficient in situations where the chance of conflicts is very small. When however more conflicts occur, the amount of work that needs to be done to resolve them grows rapidly. Much work has to be repeated when a transaction is aborted which can lead to a congested system.

Suggested literature:

[cel88] - A very formal and mathematical discussion about concurrency control. This book is originally written with distributed databases in mind, but also the principles for centralized transaction management are discussed. If one needs a formal proof about a concurrency control mechanism, it should be found here.

[cou94] - Chapter 12 and 13 of this book have been the foundation for this discussion on transaction management.

[gos91] - This book was only found during the end of the thesis study, but is a very interesting work on distributed systems. In the chapter about synchronization, transaction concurrency control is shortly discussed. More important, an entire chapter has been devoted to distributed deadlock detection.

[mul93] - Chapter 13 of the book discusses transaction-processing techniques, but in fact only gives an introduction to concurrency control. It is interesting to read because it discusses the subject from a little different point of view.

9. Distributed transaction management

In distributed computer systems data items that are used by transactions may be spread across several servers. A transaction that involves data items of different servers is called a distributed transaction. It does not matter if these data items are requested directly by the client or indirectly by servers. One of the problems we run into with distributed transactions is the atomicity property of transactions. When a distributed transaction comes to an end all servers have to either commit or abort the transaction. Furthermore by using the methods of the last chapter transactions can be serialized locally at a single server. In a distributed computer system it is needed that transactions are also serialized globally over all the servers involved.

There are two methods of executing a distributed transaction. The first is that the client makes his requests to different servers. These servers process the requests but do not make further requests to other servers. In this way however, we confront the client with the fact that the underlying computer system is distributed. The second method is that the client makes his requests to one or possibly more servers that are allowed to make further requests to other servers. In this way nested transactions are created.

When more than one server is involved in a transaction, they need to communicate with each other in order to coordinate their actions. The simplest way to structure this is by making the first server that is requested by the client the *coordinator* of the transaction. This coordinator assigns a transaction identification number to the transaction and returns this to the client. This transaction identifier can be made unique by letting it contain two parts: the (unique) server identifier and an identifier unique to that server. The coordinator also keeps a list of all servers involved in the transaction and is in the end responsible for aborting or committing the transaction.

The list of servers managed by the coordinator is maintained in the following way. Every time a client makes a request to a data item in a new server, that server is requested to join the transaction. The transaction identifier is supplied to the server, so it also knows the identity of the coordinator. The new server announces to the coordinator that he participates in the transaction, and supplies the coordinator with its own identity and the identity of the transaction it joins. In this way the coordinator knows all servers that participate in the transaction and all these servers know the coordinator, so they all will be able to find each other when it is time to commit or abort.

At the end of a transaction all of its operations must be carried out or none of them. In paragraph 9.1 will be discussed how all servers involved reach the same conclusion. In paragraph 9.2 we will discuss some methods of concurrency control to serialize concurrent distributed transactions.

9.1 Commitment protocols

At the end of a transaction all of its operations must be carried out or none of them at every server involved. The servers have to agree whether they all are going to abort or commit the transaction. Whenever a client wants to commit or abort the transaction it informs the coordinator of the transaction. When the client wants to abort the transaction the coordinator can immediately inform the servers involved to abort the transaction. If the client wants to commit the transaction, the coordinator starts the so called two-phase commitment protocol. In the first phase the coordinator informs at every server involved if it is prepared to commit or not. Every server sends back a Yes-vote if it wants to commit and a No-vote if it wants to abort. When the coordinator has collected all votes it looks if any server has voted No. If all servers have voted Yes it will send the servers the message to commit, else a message to abort. All servers will then send back an acknowledgment, so the coordinator knows when the transaction can be forgotten.

Some time-outs have to be set to be able to deal with failures. When one of the servers fails, and does not send its reply to the commit question, the coordinator has to time-out and send the abort transaction message to the other servers that did send their votes. If the server did not fail but just was to late, it will not know what has happened to the transaction. It will have to inform at the coordinator what happened. When a server gets no request to commit from the coordinator some time after it performed the last operation on behalf of the transaction, it may assume that something has gone wrong. It can then abort the transaction since no decision has been made yet. It then can perform operations on behalf of other transactions.

When there are nested transactions, the two-phase commit protocol has to be executed at every level. If one of the subtransactions is ready, it can abort or commit independently from the parent transaction. The parent transactions can based upon the result of their child transactions and votes from servers decide to commit or abort etc.

Using the two-phase commitment protocol, the servers involved in the transaction can decide together on request of the client whether or not the transaction was a success and should be committed.

9.2 Concurrency control in distributed transactions

If we look at a single server that is accessed by more than one concurrent transaction, it is still its responsibility that these transactions are processed in a serially equivalent way. This can be done by using the methods discussed in paragraph 8.2. All the servers involved in a distributed transaction are however jointly responsible for processing distributed transactions in a serially equivalent way. This means that if two concurrent transactions both access data items on more than one same server, the transactions must be processed at all those servers in the same serially equivalent order. Down here we will look how this affects the methods discussed in paragraph 8.2.

9.2.1 Locking

When locking is used for distributed transactions each server only maintains locks for its own data items. They have to remain locked until the distributed transaction has committed or aborted, so until after the atomic commitment protocol. When there exist nested transactions, the parent transactions may not be executed concurrent with their own child transactions. This can be solved by letting the child transaction inherit all the locks that are held by the parent transaction. The only problem left is that all servers set their own locks without looking to others. This means that although deadlock is prevented in the individual servers, deadlock may still happen by transactions waiting for data items locked by other transactions at other servers. How this kind of distributed deadlock can be solved is discussed later.

9.2.2 Timestamp ordering

Assigning unique timestamps in case of a single server is simple. With distributed transactions however it is needed that transactions are assigned a globally unique timestamp. The servers involved in a transaction are again responsible for serially equivalent execution of all transactions. If a pair of transactions occurs at more than one server, they should be committed in the same order everywhere. This order is determined based upon the timestamps.

A timestamp can be constructed by combining the unique server identification number and the local clock of the server. This local clocks need not be synchronized, but for efficiency reasons it is more convenient to use roughly synchronized physical clocks. When this timestamp ordering is used according to the conflict rules discussed in paragraph 8.2.3, transactions will be committed serially equivalent everywhere. Nested transactions must inherit the timestamp of the parent transaction, along with a version number to resolve conflicts between nested transactions belonging to the same parent transaction.

9.2.3 Optimistic concurrency control

A distributed transaction must be validated not only by the individual servers involved, but also by the collection of all these servers together. When more than one distributed transaction is validated at a time then it could happen that these transactions are serialized differently at different servers. A way to prevent this is by allowing only one validation at a time at every server. This however may lead to deadlock. Suppose that two transactions both update data item A at server S and data item B at server T. If they both start validation, but the first transaction starts with validation of data item A and the second with the validation of data item B, then they both can not start the validation of the other data item on the other server they updated because the other server is already busy validating.

Another way is to do a global validation after the local validations at the individual servers. The results of the local validations can be reported to the coordinator at the end of the first phase of the two phase commitment protocol. Then during the global validation it will be checked if the different orderings of the servers can be serialized. Therefore the transaction being validated may not be involved in a cycle, just like with normal deadlock.

9.3 Deadlocks in distributed transactions

Deadlocks can be detected by checking if the wait-for-graph contains any cycles, as we have seen in the case of a single server. Distributed transactions involve more than one server. It could thus happen that although there is no deadlock at any of the servers, some transactions are waiting for each other because they need access to a data item at an other server that is owned by the other transaction. This can not be seen at the local wait-for-graphs. So we have to look for cycles in the global wait-for-graph. A server could use the distributed snapshot algorithm using vector clocks discussed in chapter 5, to see if it is in a deadlock situation. Here we will discuss another method.

The global wait-for-graph can be created by combining all local wait-for-graphs. So communication is required between the servers to make the global wait-for-graph and detect cycles in it. The simplest way is to have all servers send their local wait-for-graph to a single server from time to time that checks if cycles occur. But of course we are not building a distributed computer systems to put centralized services in them so easily. A centralized deadlock detection leads to worse availability, and lack of fault tolerance; all the things we try to prevent.

A distributed method of detecting deadlock is for instance edge chasing. Here the global wait-for-graph is not entirely constructed. Servers send out probes to other servers along the edges they are waiting for. This probe can be relayed over edges the receiving is waiting for and so on. When a server receives a transaction it has sent itself, it has detected a cycle and knows he is in deadlock. He could also have detected a ghost deadlock. This happens if one of the transactions in the cycle already has aborted on the moment that the probe returns to its originator. The server thus incorrectly detects a deadlock.

Now we will look a little deeper into the edge chasing algorithm. The algorithm can be divided into three steps: initiation, detection and resolution. Now we will discuss what happens in each of these steps.

Initiation:
When a server notices that a transaction *X* is waiting for transaction *Y* to free a data item, and that transaction *Y* in his turn is waiting for another transaction to free a data item at another server, then it knows that these two might be part of a cycle. Therefore it sends a probe to the server at which transaction *Y* is blocked. In this probe the edge $X \rightarrow Y$ is represented. If there are other transactions locking the data item at the server the probe will also be sent to them because they also might be part of another cycle causing distributed deadlock.

Detection:
When a probe arrives at a server, it can be used to try to detect if deadlock has occurred. Suppose that a server receives the probe with edge $X \rightarrow Y$ then it checks if transaction *Y* is waiting for another transaction at another server. If this is not the case deadlock can not occur. If it is the case it appends this transaction to the probe and sends it to that server. When a server appends a transaction it checks if this transaction causes a cycle in the probe. If it does a deadlock situation is detected. In this way the wait-for-graph is built edge by edge.

Resolution:
Once a cycle is detected, one of the transactions in the cycle must be aborted. By giving transactions a certain priority, the choice of which transaction to abort can be regulated.

It now looks as if many messages are needed in order to detect deadlock. In general a server that wants to forward a probe, because the last transaction in the probe is also waiting for another server, asks the coordinator of this transaction the identity of the server that holds the data item the transaction is waiting for. Instead of returning the identity to the server that in turn sends the probe to the appropriate server, the coordinator could forward the probe itself. This would then require two instead of three messages. Detecting a cycle with *N* transactions requires a probe that is forwarded by *N-1* coordinators from *N-1* servers. This means that *2(N-1)* messages are needed. This is even linear with the number of transactions involved. Furthermore experience has shown that most distributed deadlocks only involve two transactions. Hence distributed deadlock detection with edge-chasing requires only two messages in this case. From all this we conclude that edge-chasing is an efficient way to detect distributed deadlocks. If there is a deadlock it will always be found. The algorithm can fail when a transaction already represented in the probe aborts meanwhile, or when a server fails, or messages are lost. This can result in the detection of a ghost deadlock.

9.4 Replicated data

Here we are going discuss the case where there are not only more than one data server, but where data items are replicated over multiple servers. The use of replicated data is discussed in chapter 7. A transactional service where physical copies of data items are replicated over replica managers is called a replicated transactional service. From the users point of view replicated transactional services appear to be executed one at a time upon one data server and upon one single data item. In addition to the earlier serializability property this is called: *one-copy serializability*. A client may send its request to each replica manager holding a copy of the required data item. The replica manager is responsible for getting the cooperation of the other managers. It can not wait for the transaction to commit before it forwards the request to the other replicamanager, although this would take less communication. The other replicamanagers need to know about the requests performed by others to be able to serialize the transactions correctly at every server.

A simple way to get one-copy serializability is by using the read one/write all replication scheme. Here a write operation must be performed at every replicamanager and a read operation at a single replica manager. This means that a write operation sets locks at every replica manager for the required data item. A read operation only sets a read lock at a single replication manager. As before two write locks conflict and a read lock conflicts with a write lock, and thus one-copy serializability is achieved. In the case of replicated data the two phase commit protocol is slightly changed. The coordinator asks the transaction servers their result, but these have to check the replication managers they accessed. When a coordinator sends a transaction commit or abort, the transaction servers have to notify their replication servers. There is added an extra level to the two-phase commit protocol.

When primary copy replication is used, the concurrency control can be done at the primary server. The primary server can wait until a transaction commits before it notifies its slaves. It is only not allowed any more to perform read operations at the slaves, because transactions may then not be serialized correctly.

Using the Read one/write all scheme above is not the entire solution. It could happen that one of the replica managers fails and recovers later. Because the replication manager is temporarily unavailable, all replication managers can not be updated. Furthermore it could happen that the network is temporarily partitioned into more than one part. What happens if not all replication managers can reach each other? In the following paragraphs we will first discuss the unavailability of the replication managers and next the partitioning of networks.

9.4.1 Unavailability replica managers

The general idea is to perform a read request at a single replica manager and to perform write operations at every available server. This is called available copies replication. This works exactly the same as with the read one/write all scheme until the group of available servers changes. When a client makes a request to a replica manager that already has failed, it times out and tries another. When a client makes a request to a replica manager that is still recovering of a failure, the replica manager will notify the client and ignore the request. It is however different when a replication manager fails during a transaction. Then the local concurrency control mechanism will not always result in one-copy serializability. This can best be shown by the following example. There is a group of replica managers that contain data item *A* and another group of managers that contain data item *B*. Now there are two concurrent transactions, the first reading *A* and then writing *B*, the other reading *B* and then writing *A*. Now both replication servers used for reading *A* and *B* fail. This means that both read locks also disappear. The write operations do suddenly not conflict anymore with the read locks, and the transactions are not serialized.

Therefore some extra concurrency control procedure has to be implemented. This procedure has to make sure that any failure or recovery of a replica manager does not appear to happen during a transaction. Before committing, a transaction checks if there were any failures or recoveries of replicamanagers of data items it accessed. If all is the same as at the time it accessed the data items, it can safely commit. In the above example if both read locks disappear before the validation of either transactions starts, both transactions will abort. If one transaction starts validating while its read lock is still present and no recoveries have occurred it can safely commit. This because no other transaction could have written a replicated version of the data item while the read lock exists. The other transaction will find its read lock disappeared (else the first transaction never could have gained the write lock) and thus is not allowed to commit.

9.4.2 Network partitions

Here we will discuss how network partitions can be dealt with. A partition is the separation of a network into multiple parts that are not able to communicate to each other due to the failure of interconnections. We will of course assume that the partition will be repaired eventually. Just like in transaction concurrency control we have here the optimistic and the pessimistic approach. In the optimistic approach updates are allowed in the partition, and after repair inconsistencies are resolved. In the pessimistic approach inconsistencies are prevented during partitions at the price of restrictions in not partitioned operation. We will now discuss some methods of overcoming partitions.

Available copies with validation:
This is a optimistic approach. It maintains the available copies replication method in every partition. When a partition is repaired a validation is needed to see if transactions in different partitions have conflicted, and produced inconsistencies. When the validation fails, one of the conflicting transactions has to abort after all. This means that it must be possible to reverse the actions performed by that transaction. This method can thus only be used in environments where this is possible. Validation can be performed by having each partition maintain a log of the data items affected by read and write operations of

transactions. This log will not show conflicts because concurrency control has been applied within the section, but when a partition is repaired, the logs can be linked and checked for conflicts.

Quorum consensus method:
Conflicts between partitions can also be prevented by allowing only one of the partitions to carry out operations. A subgroup of managers whose size gives it the right to perform operations is called a *quorum*. The replication managers that do not belong to the quorum thus have out of date copies. With the help of timestamps or version numbers only the most recent copy will be used.

Another way of determining a quorum is by votes. When a operation is performed first votes are gathered for that data item. Every group of replica managers must obtain a read quorum of R votes for a read operation, and a write quorum of W votes for a write operation. R and W must be such that W is greater than half the total votes and $R+W$ must be greater than the total number of votes for the group. When a read operation is performed first more than R votes are gathered. From the definition of R and W it follows that every read quorum overlaps with every write quorum ($R+W>$total number of votes). So at least one up to date data item is contained in the quorum, that can be identified by timestamp or version number. Write operations are performed when a write quorum can be reached on all replica managers in the quorum. The version numbers of the new data items are also increased. When a partition is repaired, older versions can be replaced by new ones from the other partition.

Virtual partition:
The quorum consensus method is expensive for read operations compared to the available copies method. Therefore the two methods are combined. A virtual partition is a group of replication managers that is big enough to have a read and write quorum. On this group the method of available copies can be used, with the advantage that a read operation only has to be performed at one manager. Whenever the group loses its read write quorum it tries obtain a new virtual partition. When this happens during a transaction, it must be aborted. After a new virtual partition has been created, all data items must be updated, because read operations are performed only at one replication manager.

9.5 Conclusion

The concept of transactions is an important one. There exist many single actions in a network that are composed out of multiple operations reading or writing data items. Transactions also help us to maintain data consistency, if they are shared by different processes. As we have seen it is even not easy to perform transactions on a single data server, at least not when concurrency is allowed. The concurrency control mechanism that is used most is locking. It is rather simple to implement, but has the danger of deadlock. Deadlock can be detected or prevented well by the methods we discussed.

In distributed computer systems, where there are more dataservers, distributed transactions impose their own problems. Locally the same concurrency control can be used. Transactions at one server do however not necessarily have to know about transactions at others. The servers have to cooperate to allow concurrency, but prevent inconsistencies. Extensions of the methods for local concurrency can be used for global transaction control. Communication between the different data servers is important to maintain this global state of consistency. The atomic commitment protocols are used for this communication. Final issue was the use of replicated data in distributed transactions. This means that the unavailability of a single server, or even a network partition can impose serious problems on the up-to-dateness of data items. We have discussed some methods for dealing with this.

From this we can conclude that transactions in distributed systems are possible. However, to allow concurrency a lot of effort has to be put in the design of the system. By using a network wide strategy or protocol, and cooperation of the servers involved good transaction management can be created.

Suggested literature:

[cel88] - This book performs in-depth research of transaction concurrency control. The different concurrency control mechanisms are discussed in a very formal way. So if one needs proof of what is discussed above, this is certainly the book.

[cou94] - Chapters 12, 13 and 14 have formed the base for the chapters on transaction management.

10. Global naming

In a distributed computer system, names are used to identify numerous objects like for instance: files, users, computers and services. Names are very important since they are the key to sharing resources between different processes and they are needed to communicate between different servers or users in the system. If for instance different processes want to share the same resource, they must both use the same global name so the right resource is identified in the system by both of them. Furthermore users or processes cannot communicate with each other if they cannot name each other. The name service can be seen as a central point for other servers and their clients, because the name service enables other services to identify and access objects in a uniform way.

A name is a string of symbols that identifies an object. In distributed systems only a name is not enough, because also the current location, or address, of the object must be known. So a name specifies what a process seeks, and an address specifies where it is. If necessary the concept of a route can be used to specify how to get there. In a distributed system the naming facility should be seen as a global space of identified objects, rather than a space of identified host computers containing locally identified objects. It should be independent of the physical topology of the system or the current location of the named object. Objects also must be movable without the user noticing it, hence without changing name. It should provide meaningful names to the human users, but also convenient identifiers for computers, so some mapping between these two levels must be provided.

The connection between a name and an object we will call a binding. Naming can be structured in different ways. So called flat names are names used as an unique identifier within the domain without internal structure, that is always the same for the particular resource it is identifying. This is done to make the name location independent. A partitioned name is structured as a series of primitive names identifying, respectively, a domain, a subdomain, a sub-subdomain etc. Hierarchical names thus also belong to the partitioned names. Names are always associated with some context, which can be defined as the environment in which a name is valid. In a distributed computer system, contexts often represent a partitioning of the name space. For example, in the partitioned name A/B/C, B/C is a name explicitly existing in the context of A. This partitioning often happens along geographical or organizational boundaries. Note that the partitioned name is very useful here.

If we want to have the freedom to place the object on another place and several processes must be able to share it then there must exist a mechanism to resolve the name into an address where the processes can find the physical copy of the resource. This mechanism could be a naming service consisting of a table with all names and the location of the resource it identifies. For a large number of names this is not a workable situation. In a distributed system the lookup tables are even replicated and distributed over the network, so it is not clear where the name can be looked up.

Thus what we need is a naming service that maintains a database that stores the bindings between textual names used by users and attributes of objects. The service also must resolve the names. In the next paragraph we will discuss a model of a name service for distributed systems.

10.1 Name services

A name service keeps a database of bindings between a set of textual names that have a meaning to human users and attributes of the objects. An important task of the name service is to resolve the names. Among the attributes are also addresses of the objects, or information about where to find the next piece of information for resolving the name. This database of course does not consist of a single component in a distributed computer system. It is often replicated and spread across the network. This to enhance availability and fault tolerance and to bring the service geographically closer to all the users.

So the main difference between naming in distributed systems and in centralized systems is that in a distributed system there is no place that has to have the information about all named objects in the system. The items of concern are managing the replicated name services across the network and making sure that objects are named consistently so processes can communicate about the object and share it. Also when an object is moved, or deleted the location of the object must be found from every part of the (replicated) name service. Since this is in fact the management of a replicated database, the techniques from earlier chapters involving consistency and replication management can be used here. Furthermore the name service must be scaleable. When new domains are added to the network, the naming that is used there must be merged with the existing naming scheme. This could lead to conflicts between names, or worse, the other naming scheme might be totally differently structured.

To model the name service, we will use the concept of contexts as discussed previously. These contexts are linked in a hierarchical way. As a result the naming space is also hierarchical, and partitioned names will be used. The contexts are presented as naming domains in which exists an administrative authority that is in control of the binding of names in its domain. This administrative authority can consist of one or more name servers responsible for the lookup and management of bindings of names in the context of the domain. If the object is not located within that domain, the name contains an attribute that contains information where to find the next naming domain. The location of name servers should be transparent to users. Clients therefore make requests to the name server through the use of a name agent. The functions performed by name agents are speaking the proper communication protocol, and maintaining a detailed knowledge of the name space and of existing name servers. Using this model the naming service is totally location transparent to the user. The name itself is also independent of the location of the object.

The hierarchical name thus says something about the path along which it is resolved. When for instance the name *root/name#3/name#5/objectid* is resolved (Figure 11), the name *name#3/name#5/objectid* belongs to the context of the *root* naming domain. A name server in the *root* naming domain contains one or more addresses of name servers in naming domain *name#3* as an attribute. The name server in *name#3* in turn contains the address of a name server in *name#5* where *objectid* is located.

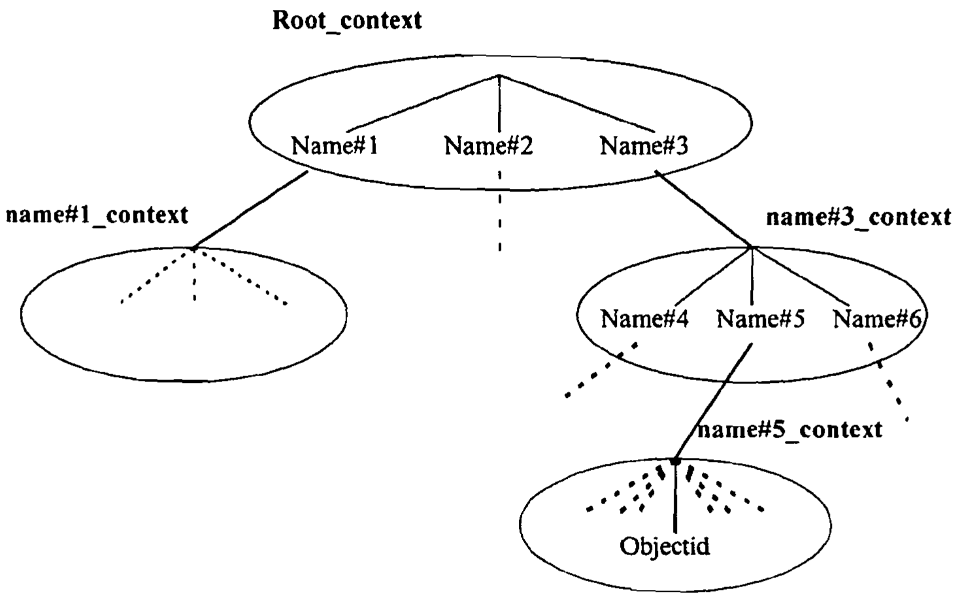


Figure 11 Hierarchical name space

An improvement to name resolution can be reached by caching. Since naming data in a computer system does not change very frequently the attributes produced by the name resolution can be cached at the requesting server. The server does not have to request the name service to look up the name next time. Even new names can be resolved more quickly if the first part of the name is the same, and thus their attributes are known. It also protects against failures. If one of the lower level naming servers is down, but the attributes of its children are cached, the name can still be resolved because these children can be accessed directly. From there normal name resolution can be used. When caching is used, out of date attributes may be used. The requesting client will however notice, because no service accepts the request or errors are returned. Normal name resolution can be started then.

Another advantage of a hierarchic naming space is that it is the only known naming space that scales well. When a new naming domain is added, it can be added to the existing naming domain as a subtree. The added naming domain can still be managed in the way it used to be without conflicting with the existing one. Global names can be resolved upto the root of the subtree, where this root takes over the rest. This also means that the subtree uses its own naming service. Furthermore users can select names autonomously without conflicting with other objects managed by other services. Because they are managed by different services the global name will have a different prefix, because the name eventually has to resolve into the location of that service. If a name is chosen that already exists for an object managed by that same service, the service will notice and notify the user.

10.2 Conclusion

To allow scaleability and an efficient way of name resolving a hierarchic naming space should be used. With the assumption that names do not change frequently, the performance of name resolution can be enhanced by caching at the name servers. To enhance availability and reliability, the naming service can be replicated and distributed using the techniques from chapter 7.

One of the disadvantages of the naming service as we have discussed it, is that all names are absolute. That is they all start at the root. Names can grow very large in this way, and hard to memorize. This can be solved by creating a working context or aliases for a client. When using names within the context the entire global name is not needed, but stays of course valid. This aliasing is done at the client, so the complete name still is presented to the name service.

A more disturbing problem arises when the naming service should be restructured. Let us look at a situation at an University. When the department IT from the faculty E would become a faculty itself, the name server structure would ideally become uni.fac.IT instead of uni.fac.E.IT. This would however mean that all names in the system should be changed which is not all that easy. An alias for the new location could be created when the old name is used, but when this happens more often the naming service will be overloaded with all sorts of crosslinks. Here we take the consequences of not having names as unique identifiers, but having names that also say something about the structure of the name service. When the naming service is structured as an image of the organization where the distributed system is used, like the university example above, restructuring the organization or naming service can lead to awkward situations. As we have seen a naming service is an important part of a distributed system, It should be thought over well, before implementing it in a specific situation (or organization).

Suggested literature:

[cou94] - Chapter 9 of this book has formed the foundation of the discussion above, but has been extended using the discussion in [gos91] below.

[gos91] - This book discusses the subject of naming in distributed systems in great detail. It has been used to refine the contents of this chapter that was originally based upon [cou94]. People who would like to know more on the subject should read chapter 7 of the book.

[mul93] - Chapter 12 provides only provides a short introduction, along with a discussion of two known name services.

11. Global file service

Every user in a (distributed) computer system produces and uses data. There must be a way to store this data permanently. In a distributed computer system it must be possible to distribute the data. Data can be replicated to increase availability and reliability. An user may wish to access remote data, or even may wish to access his or her own data from a remote location. Data is also often used for sharing between different users or processes. Because data must be accessible from everywhere in the computer system, it must have a globally unique name. Data can also be copied or moved across the network. This means that data access must be location independent.

The most common used method for storing data is by using files. A file is usually a sequence of data items (bytes) stored permanently somewhere, and accessible by read and write operations. Because of the great importance of files in conventional computer systems, special file system services are added that are responsible for the management, retrieval, storage, naming and sharing of files. In a distributed computer system, the fileservice is even more essential. It performs the same tasks as in conventional centralized computer systems, but is also responsible for maintaining global properties in a distributed computer system.

11.1 Distributed file system

A distributed file system is an important component of a distributed computer system. It has to fulfill the same functions as in centralized computer systems, such as naming, storage and retrieval of files. In a distributed computer system however, some extra functionality is also required. It must enable user programs to access remote files, without the need of copying it to the local permanent storage. Users at locations without permanent storage need to be able to access remote files. It also must provide the sharing of (remote) files between multiple user processes. The files that are replicated across the network need to be managed so they can be accessed in a consistent way by different user processes. Furthermore the naming of the files is important. Every user process must see the same global name space for files. This means that files always keep the same name until they are removed from the system, although they may be copied or moved from one physical location to another. The distributed file system must provide location independence.

As we can see a distributed file server contributes to the global properties of a distributed computer system. The user should see the entire system as if he were sitting behind his own computer, accessing all data locally. The user is unaware of the physical location of files, because he uses names from the global file name space. The file system makes sure that the physical location of the file will be retrieved and the file will be accessed either locally or remotely completely transparent to the user.

In short the following issues should be addressed in a distributed file system:

Access transparency:
A client program should not notice any difference when accessing a remote or a local file. The same set of operations for accessing both remote and local files should be used. In this way programs written to use local files, can use remote files as well.

Location transparency:
As we have said above, the client programs should be unaware of the physical location of the file. An uniform global name space thus should be provided. This also means that files can be physically relocated without changing the name, so again the client program is unaware.

Concurrency transparency:
Concurrent operations of different client programs upon the same file must be executed in a consistent way.

Replica transparency:
Files may be replicated across the system to enhance availability and fault tolerance, without client programs noticing it.

Consistent naming:
Every client program uses the same text name from anywhere in the distributed system for accessing a file.

The issues above are the most important in order to achieve the global properties of a distributed computer system. Other issues that are important when implementing a file system for a distributed system are security and that it must be scaleable to allow growth of the system. The issues listed above must sound familiar by now. We have discussed them already in some form in the previous chapters of this thesis. In the next chapter we will give a general description of a file system. For achieving the properties listed above, we will rely heavily on previous chapters.

11.2 A general model

In a centralized operating system, the following layered basic services are provided by the file system:

- **Disk service layer:**
  This service performs disk I/O and buffering, and accesses and allocates disk blocks.

- **File service layer:**
  This service layer relates file identifiers to files, and reads or writes file data or attributes.

- **Directory service layer:**
  This top level layer provides the required mapping between file text names and file identifiers used by the file system.

Remember that we have defined a file as a sequence of data items stored at a permanent storage. The file is often physically divided on disk into blocks. So the file service layer can also ask for certain blocks of the file to be read. The naming of files within the file system is done by using directories. A directory is a file itself, that contains the mapping of the textual names known to the outside world and the file identifications used inside the file system. A directory itself may contain other directories, what thus will result in a hierarchical structure.

A distributed file system should provide the same services as the centralized file system. Additional issues and services should however be considered. To obtain location transparency we need to have some sort of global naming. In the last chapter we already discussed this subject. There files belonged to the objects that are named in a distributed computer system. Files can also be replicated and some transaction management is needed because client programs may try to access a file simultaneously. Let us look into how this could help us to develop a general model for a distributed file system.

- **Disk service:**
  The disk service is concerned with reading and writing raw blocks. It provides a logical view of a disk storage to the layer above. It is often separated from the file service layer, so different storage media or methods of storage can be combined.

- **File service layer:**
  The file service layer provides the layers above with the abstraction of files. Possible operations that can be performed on files are: opening and closing files, writing and reading files starting from some particular position. It also maintains the attributes belonging to the files, such as for instance date of creation and the filelength.

- **Transaction service:**
  In a distributed computer system it happens that multiple concurrent user programs want to perform operations on the same file. These operations by an user program can be seen as part of a transaction. How concurrency can be controlled for these transactions is discussed in chapters 8 and 9.

- **Replication service:**
  If there is more than one replica of the file also replica management has to be performed. The methods discussed in chapter 7 can for instance be used for this.

- **Naming/directory service:**
  This is the service that maps the textual names to the unique file identifiers used by the file system. In a distributed system there could be more than one file server, but the client should not be aware of that. All servers must collaborate to provide a file service that is transparent to location, distribution and replication of files. Global naming as discussed in chapter 10 can be used for this. Here also access control is done. When a file is requested by an user, it is checked if he has the right to. Also the sharing of files by more than one user program can be managed here.

The last part that is required is a client interface at every computer. This client interface offers a set of operations that is similar to input and output operations provided by the operating system of the computer. All file access from and to the directory service is thus translated, to make the file access transparent to the user. This interface can also be used to cache files to improve performance. If there exists a local instance of the file system, then it is checked first to see if the file is stored locally.

This all leads to the general model depicted in Figure 12.

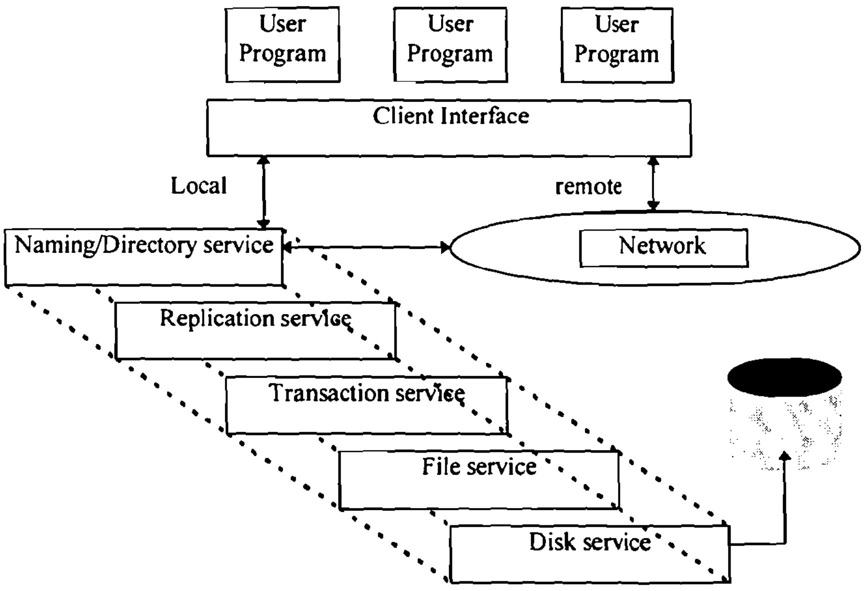


Figure 12 General model of a distributed file system

In the model above, location transparency is already offered by dividing the basic file service and the directory service. Note however that the directory service also fits quite easily into the global naming model presented in the last chapter. In the global naming model a name server keeps the location of the next server that contains more information for resolving the name. This eventually leads to the service that manages the requested object. In case the object is a file this would be the directory service. The directory service can in fact be seen as an specific name service, since it maps textual names onto file identifiers.

11.3 Conclusion

In this chapter no detailed model of a distributed file server is given. Some properties that a distributed file server should own are mentioned. The most important issue is transparency. Users should not be aware of the physical location of a file, they should all see the same file name space and they should not notice whether they are accessing a remote or a local file. A general model is given to show some of the basic concepts. By using the methods discussed in previous chapters the desired properties can be achieved somehow.

In practice many kinds of file services have been developed, all approaching the problem from a different point of view. In these implementations mainly some of the properties mentioned above are addressed. It is not easy to take all the desired properties into account, and still produce an acceptable file service. This because we have to keep in mind that the file service is the most heavily used service in a computer system. In order to gain performance some concessions then have to be made.

Suggested literature:

[cou94] - The discussion in chapter 7 of this book has been used to write this chapter. Chapter 8 additionally contains case studies of existing file systems.

[gos91] - Chapter 13 of this book has been used to refine the discussion on the subject of file systems. Especially the model for a distributed file system presented there, has been used in this chapter aswell.

[mul93] - Chapter 14 discusses the subject of distributed file systems. It is not a very detailed discussion, but serves well as an introduction. Two distributed file systems are reviewed in more detail.

Part III

In this part the operating system Windows NT will be studied in order to determine if it may be called a distributed operating system according to our definition in part I of this thesis. It is developed by the software company Microsoft and has been officially released early 1993. Windows NT is chosen because it is commercially available to the public and, although it has not been designed for the sole purpose of building a distributed system, it does claim to be the future in distributed computing.

12. Windows NT

Here we want to examine a commercially available operating system to see if it also may be called a distributed operating system according to our definition in part I of this thesis. There are many operating systems that have been designed to build a distributed computer system. Most of these operating systems are however developed for the purpose of research of distributed systems at universities or laboratories. Only few of these are commercially available to the public. If they are designed for the purpose of research in the area of distributed computing, relatively much information and results are published. If an operating system is however developed by a commercial company this is not the case.

We have chosen to examine a new operating system *'Windows NT'* developed by the software company *Microsoft* and officially released early 1993. This choice has been made for a few reasons. First of all, Windows NT is commercially available to practically the entire world (if Microsoft is not commercial, then who is?). It has also been released quite recently and thus presumably built for the near future in computing. Because it has been released only recently and it has been developed by a commercial company, only little research on Windows NT performed by others than Microsoft is known. Although Windows NT has not been designed for the sole purpose of building a distributed system, it does claim to be the future in distributed computing [cus92]. That makes it all the more interesting to see whether wc think they are right or wrong. A disadvantage of examining a commercial product however is that real inside information is hard to come by.

12.1 Structure of Windows NT

On the outside Windows NT looks very much the same as its predecessor Windows 3.x. Windows NT is internally however structured in a very different way. It was built from the ground up and therefore named NT which is an acronym for 'New Technology'. Some of the design goals set by the Windows NT developers include:

- **Extensibility** - the code must be able to grow when market requirements change.
- **Portability** - the operating system must run on different processor types.
- **Reliability and robustness** - the system should protect itself from internal failures and failures from the outside.
- **Compatibility** - its user interface and application programming interfaces (API) must be compatible with other (Microsoft) systems.
- **Multiprocessing** - applications must be able to take advantage of multiple processors when available.
- **Distributed computing** - it should be able to distribute its computing tasks to other computers on the network. Local and remote computers should be employed without user intervention.

- **Security** - It should be compliant to security guidelines of the US government.
- **Performance** - the system should of course be as fast and responsive as possible.

Just like in the general model discussed in chapter 2, Windows NT is based upon a micro-kernel architecture. Windows NT can be divided in two parts: the user-mode portion that consists of the Windows NT servers, and the kernel-mode portion called the executive. These Windows NT servers are also called protected subsystems. Each of them resides in a separate process whose memory is protected from other processes. The executive is the operating system engine that can support any number of these server processes. It provides the system functions for low-level processing for the protected subsystems. The total structure of Windows NT is schematically depicted in Figure 13. In the following paragraph we will discuss the executive and the protected subsystems in more detail.

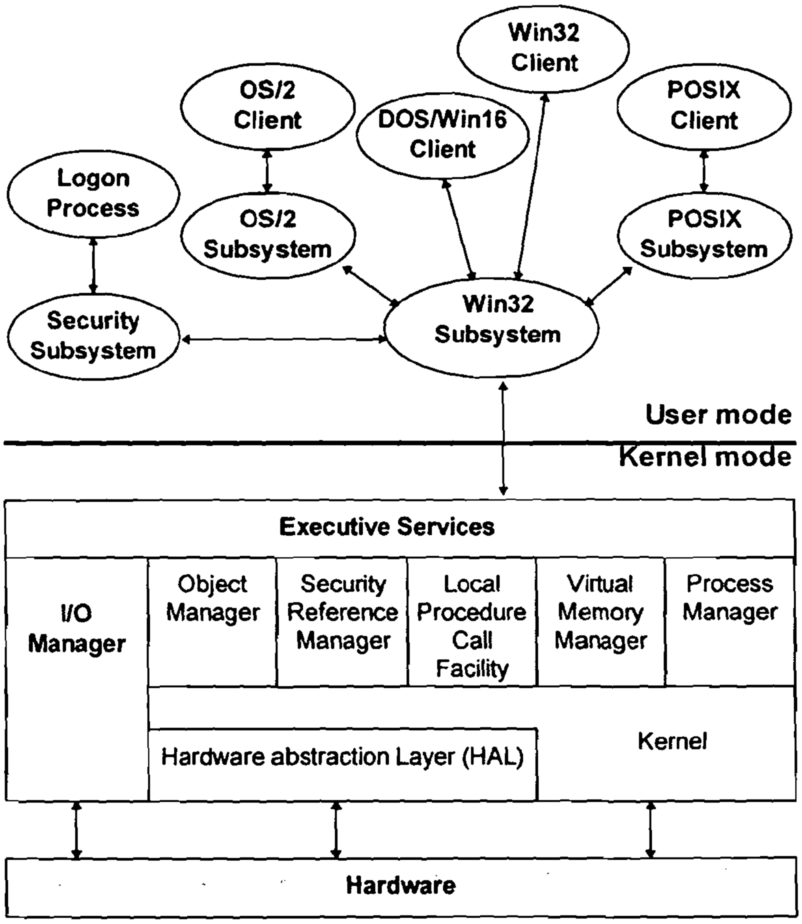


Figure 13 Structure of Windows NT

12.2 Protected subsystems

Some of the design goals mentioned above are compatibility and extensibility. Popular applications that nowadays are used on existing and established operating systems should also run on Windows NT. If this were not the case, users would not give Windows NT a second look. Furthermore Windows NT should directly support the popular Windows graphical user interface. This all is solved by using protected subsystems in user mode. These are called protected subsystems because each resides in a separate process whose memory is protected from other processes. If they desire to communicate then this must be done by passing messages. The concept of processes in Windows NT will be discussed later.

The most important subsystem is Win32. First of all it supplies the 32-bit Windows Application Program Interfaces (API) for 32-bits windows based application programs. Furthermore it provides the graphical user interface and it controls all I/O from and to applications. The 'old' 16-bits windows 3.x and DOS applications do not have their own protected subsystem. These applications are serviced by a Win32 based application called a virtual DOS machine (VDM) that emulates a complete MS-DOS environment. Here Windows 3.x is seen as a single MS-DOS application running in a VDM.

Windows NT also provides compatibility with OS/2 and POSIX by respectively an OS/2 and POSIX subsystem. These subsystems provide API's for their specific application programs. OS/2 is an operating system developed by IBM while POSIX (Portable Operating System for Interactive eXecutives) is a procurement standard for government computing contracts specified by the US government. Because the Win32 subsystem handles all video output and keyboard input, the other subsystems are clients of the Win32 subsystem. Note that this part of Window NT is designed using the client/server model. When an application or another server calls an API, a message is sent to the server providing that API. This message eventually is sent through the executive, but for ease of reading this is not shown in Figure 13. The server then replies by sending a message back to the caller. When an user runs an application the Win32 subsystem does not recognize it calls another subsystem to run it, or it initializes a VDM in case of a DOS or Win16 application.

Using this design Windows NT can relatively easy be extended with other API's to provide compatibility for applications from other types of operating systems. This is done by designing a protected subsystem that in turn will provide the API's for these applications. Using the client/model on the top level of Windows NT is therefore a good choice for providing compatibility and extensibility.

12.3 The executive

The executive consists of a number of components. They all implement two kinds of functions: *system services* that can be called by the subsystems and other executive components, and *internal functions* that are only available to components in the executive. Unlike the protected subsystems, the executive does not run in a process of its own. When important system events occur it takes over the execution of a process, and thus runs in the context of that process. Hence it is constantly providing system services to the running processes. The components are independent of each other. The interfaces between them are carefully described, so that a component can be replaced as long as the new component complies to the rules. We will now in short discuss the different components of the executive.

Object manager:
It creates, manages and deletes executive objects. An object is a data structure whose physical format is hidden behind a type definition, a set of services that may be requested on it, and a set of formal properties the object possesses. In an operating system the data hidden in objects are for instance: files, processes, blocks of memory etc. The advantage of using objects is that system resources are accessed uniformly. Furthermore objects are easier to share between two or more processes. Processes only need to open a so called handle to the same object.

Process manager:
It creates and terminates processes and threads. It also suspends and resumes the execution of threads. We will examine processes and threads more closely later.

Local procedure call (LPC) facility:
This facility is responsible for the passing of messages between a client process and a server process on the same computer. The LPC is a specially for this purpose optimized version of the Remote procedure call discussed in chapter 6. Here optimized means simplified and therefore the LPC protocol is also called the Lightweight RPC protocol.

Virtual memory manager (VMM):
It provides a private (virtual) 32-bit address space of 4 gigabyte to each process, and protects these address spaces from each other. It also manages the paging of memory to disk when memory usage is too high. For an extensive discussion of memory management in Windows NT see [yao92].

Security:
The security monitor takes care of the security on the local computer. It guards the operating system resources by performing object protection and auditing. This is another advantage of the use of objects. All objects are protected in the same way, which simplifies security. Although security is an important issue in Windows NT it will not be discussed here.

Kernel:
This is the real microkernel part that performs the real low level communication with the underlying machine. It responds to interrupts and exceptions encountered. It is also responsible for scheduling the different threads of execution, and synchronizing the

activities of multiple processors at the same machine. It also supplies lower-level interfaces and objects used by the higher-level parts of the executive.

I/O manager:
The I/O manager has been structured in a very layered fashion. Applications do not access the device drivers directly, they communicate with the I/O manager which communicates with the device drivers on their behalf. Because the application only see the I/O manager, it provides device independent input and output facilities via the executive. This includes Windows NT drivers that accept file requests and translate these into I/O requests for a particular device, but also drivers that provide intermediate I/O processing between a high-level driver and a specific device driver. These low-level device drivers that directly manipulate hardware for I/O from and to a physical device, are also implemented here. To improve performance of the file system a cache manager is also included in the I/O manager.

Hardware abstraction layer (HAL):
The HAL is often modeled as a part of the executive. In fact it is a layer of code between the executive and the hardware platform on which Windows NT is running. It hides hardware dependent details of for instance I/O interfaces, interrupt controllers, multiprocessor communication.

The higher-level portion of the executive is built out of different parts which communicate through a well defined interface. The lower-level portion and the I/O manager are however built in a more layered fashion. This is done to make Windows NT better portable. Processor specific code is located in the lowest layers of the kernel, and thus processor differences are hidden from the rest of the operating system. Some hardware specific code is also found in the I/O manager in low level device drivers. Platform dependent code is located in the HAL. A computer manufacturer can write a HAL for his own product, and thus make sure that Windows NT will run on it. For a heterogeneous set of hardware, only the lowest parts of the executives have to be adapted in order to run Windows NT. Device drivers and higher level routines avoid processor- and platform-dependent code by calling kernel and HAL routines.

Suggested literature:

[cus92] - A general overview of the design goals established for Windows NT and a discussion of the general structure of Windows NT. This article has served as starting point for this chapter.

[lin93] - A product review from a magazine. No in-depth discussion, but great for an overview of the possibilities provided by Windows NT.

[ric94] - This book discusses Windows NT in a more profound way. Although the general structure is discussed, the book also elaborates upon how Windows NT should be programmed.

[rul94] - This book mainly discusses networking Windows NT. The first chapter however presents a general introduction to Windows NT.

[ude92] - This is also a product review from a computer magazine. Next to the general overview of Windows NT it also contains a more technical in-depth study of Windows NT. Great for getting a first impression of Windows NT.

13. Windows NT processing

In the previous chapter we already mentioned processes and threads. These are the main concepts in Windows NT processing. A process is a running instance of an application and consists of blocks of code and data that are located in their own protected 4 gigabyte address space. A process also owns other resources such as files, threads or other objects. If processes do want to share memory they have to use memory mapped files because they can never access one another's address space. This mechanism is extensively discussed in [ric93a]. A thread is the unit of execution in a process and is associated with a sequence of CPU instructions, a set of CPU registers and a stack. Thus a process does not execute code, it is only the address space where the code is located. Unlike 16-bit windows a process in Windows NT can contain several threads. These threads are what Windows NT schedules CPU time to. The different threads are created, destroyed and scheduled at the kernel layer in the executive. Processes and threads themselves are managed as objects.

A process is created by loading the execution code into its private virtual address space. When it is created its first (primary) thread is automatically created by the system. This thread can create more threads which in turn can create their own etc. It can of course be very useful to have multiple threads within a process. Take for instance a spreadsheet application. The primary thread can handle all typing by the user, while another thread performs recalculation when the user is not typing. A process dies as soon as all its threads have died. All ownership of resources that has not yet been released by the process itself is destroyed, and the resources are then reclaimed by the system. Finally the virtual memory space is returned to the system. Now let us look into how the different threads are scheduled

13.1 Thread scheduling

Windows NT schedules time slices to all the active threads. Threads can also be suspended, for instance because they expect a message but their message queue is empty or because they are waiting for an event. A thread becomes active again when the message arrives, or the event occurs. If the machine contains more than one CPU, every CPU is assigned a thread that is not suspended and waiting to be processed. The execution of threads on multiple local CPU's is called symmetric multiprocessing (SMP), because the underlying hardware must be symmetric. In asymmetric multiprocessing (ASMP) one processor is generally selected to run operating system code while the other processors are used to run application code. ASMP is however difficult to make portable, because the operating system must be rewritten substantially to support asymmetric hardware. SMP systems do allow the operating system to run on every processor because it will always see the same environment. In this way physical memory can also be shared, and the operating system activities are less likely to become the bottleneck. Remember that the address space between processes is never shared. Because with SMP processes (including operating system processes) can be scheduled uniformly over the processors this method is chosen in Windows NT.

Threads are also assigned priorities by the system. Threads with the highest priorities are schedules first. If there are more threads with the same priority they are all assigned to a different local CPU, and if this is not possible all are assigned an equal amount of time in a round robin fashion. When a thread awakens that has a higher priority than a running thread, then the running thread is preempted and put to sleep until there are no more active higher priority threads. It would seem that low priority threads are never executed, but most threads spend their time sleeping allowing other threads to run. Some are merely executed more often than others. This ability to interrupt a thread at almost any time to favor another more important thread is called preemptive multitasking. It makes sure that the system is very responsive to important events because applications can be forced to give up time, unlike Windows 3.x's cooperative multitasking. Fore a more in-depth discussion about thread scheduling see [ric93b].

13.2 Synchronizing threads

In multithreaded environments synchronizing threads is very important. The access to data and system resources must be serialized to prevent inconsistencies. Furthermore when several threads have a causal relation, they must have some way to signal each other when they are ready. If for example a tread reads a portion of a file and another threads analyses this portion, then the first threads needs to be signaled when to read a new portion, and the other thread needs to know when it has finished reading. Windows NT does not do this automatically but the executive provides several synchronization objects a programmer can use. These objects are: *critical sections*, *mutexes*, *semaphores* and *events*. All but one of these objects can be used to synchronize threads across different processes.

A Windows NT object can be either *signaled* or *not signaled.* Threads and processes can be put to sleep until one or more defined objects become signaled. Processes and threads are objects themselves and only become signaled as soon as they terminate. If a parent process thus has to wait until a child process finishes it can put itself to sleep until the child process becomes signaled. The first object we will discuss, and that can only be used to synchronize threads within a single process, is the critical section. A critical section can be used to allow only one thread at a time to gain access to a region of data. If a thread has gained access to the critical section, no other thread can access it anymore. In fact this is just a form of locking. The programmer himself has to prevent deadlock, for instance by accessing critical sections in the same order at every thread.

If we want to synchronize threads with other events occurring in the machine or with operations in other processes we need the other objects designed especially for synchronization. These are the event, mutex and semaphore objects. When process or thread objects terminate they become signaled and they stay that way. With the synchronization objects this is not true. When they are owned by a thread then they are not signaled, and when they are not owned (anymore) then they become signaled so that waiting sleeping threads can be wakened and gain possession of the object. If a thread is waiting for multiple objects then it can only gain access to these objects if all of them are signaled. This to prevent the occurrence of deadlock.

mutexes:
Mutexes are very much the same as critical sections, except that they can be used to synchronize data between processes. Only one thread is allowed to own a mutex. If another thread tries to access the mutex it is put to sleep until the mutex becomes signaled again. The process that creates the mutex is the first owner. If another process wants to access it, it has to open a so called handle to it. This other process needs to know the name of the mutex, so when two processes are going to use the same mutex the programmers have to agree on a name at the design. Another method where no name is needed is duplicating the handle by the process that created it and assigning it to another process. The creating process thus has to know that another process also needs a handle. Details of this mechanism are discussed in [ric93c].

Semaphores:
Semaphores are much alike mutexes. The main differences are that Windows NT does not keep track of which thread owns a semaphore, and that a semaphore contains a reference count. Semaphores thus allow multiple threads to gain access simultaneously. A semaphore is signaled when its reference is greater than zero, and is not signaled when the reference count is equal to zero. When a thread tries to access a semaphore with a zero reference count, it is put to sleep until another thread releases the semaphore.

events:
An event object becomes signaled if a predefined situation occurs. Ownership is not important, the event object is triggered when something interesting occurs for one or more certain threads. In this way an event can for instance alert one or more threads after an I/O completion.

Thus the windows NT executive provides objects for programmers to serialize their processes at a single machine. Concurrent programming has to be dealt with by the programmers themselves with great care. Windows NT only gives the tools but does not do it automatically. Because threads are managed centrally at the executive, it knows all about the processes that are executed on that machine. Things as logical clocks discussed in chapter 5 are thus not needed here. Distribution of processes to remote machines is not provided by the Windows NT executive. We will later discuss how remote processes can be performed in the higher levels of Windows NT.

Suggested literature:

[ric93b] - This article discusses the concept of processes and threads and how they exactly are created, managed and destroyed. This article is very practical as it provides programming examples, and presents the basic functions needed within Windows NT.

[ric93c] - This article is a continuation of the article above. Here synchronization techniques are discussed, again in a very practical way. Examples are present to illustrate how applications can use the synchronization objects.

[ric94] - Chapter one of the book is clearly derived from his earlier article [ric93b], while chapter five is a derivation of [ric93c]. The book however places the subject more into perspective compared to the rest of Windows NT.

14. Networking Windows NT

Windows NT is designed with networking in mind. Every Windows NT machine provides networking features. First of all it offers peer to peer networking in workgroups as in for instance Windows for Workgroups. Furthermore, every Windows NT system is a fully functional file server capable of supporting an unlimited number of network clients. It provides file and printer sharing to an unlimited amount of users. There even exists a more extended version of Windows NT called *Windows NT advanced server* that provides domainwide administration.

The problem with the peer to peer networking is that every Windows NT machine maintains its own database of user accounts. If we want to share for instance a file with somebody else on the network then we can set the access rights to the directory containing that file to full access. This means however that everybody on the network has access to that file, because there is no central administration. The only solution is to give that person an account on our machine, and give that person access rights in the local user account database. Windows NT advanced server introduces the concept of a domain. A domain is the group of NT workstations that are part of a Windows NT advanced server network. An advanced server can be designated as domain controller, that maintains a central user account database that applies to all servers within the domain. An user only needs one account to gain privileges for the entire domain [rul94]. More than one domain controller may be present for the same domain. The second domain controller then serves as a backup in case the primary domain controller fails. Hence user account information is replicated using a primary backup replication scheme.

In Windows NT it is possible to share some resources. As mentioned above it is possible to share files, but also directories can be shared across the network. Also printers and cd-rom drives can be shared by multiple users within a workgroup, or using a Windows NT advanced server within a domain.

Another feature of Windows NT advanced server networking is that directories and files can be specified for automatic replication. The duplicated directories and files are continuously updated when the original directories and files are changed on the server system. Updates need to be done at the server while read requests may be sent to every workstation.

The built in networking features are built to Microsoft's LAN manager specifications. Microsoft does however not have a monopoly in the networking business, so also other networking protocols are supported. This is relatively easy, because of the layered device driver model of the I/O manager. At the device driver level an interface is defined as how network device drivers should be built. A network provider then has to write an installable file system to provide services on that network and a transport driver for the type of networking protocol in use. In this way also the frequently used TCP/IP networking protocol is built in, in addition to the native LAN manager compatible networking.

Suggested literature:

[rul94] - An interesting book for people who are interested in networking Windows NT. It contains a lot of information about the networking possibilities of Windows NT.

[usi93] - This very complete user guide discusses in chapter 25-31 how workgroups and domains are created, directories and printers are shared etc. It is mainly an users guide, but it also provides some background information.

15. Remote processing

We have already discussed the processing locally at a windows NT machine. We have also seen that Windows NT is designed for networking. We will now discuss if it is possible to distribute processing across the network. On the same computer messages between client processes and server processes are passed using the local procedure call facility. This is an optimized form of the RPC, a communication mechanism for executing processes on a remote computer. The Windows NT executive only performs local process management, but at the application level a RPC interface toolkit is standard included in Windows NT. The RPC interface also has the ability to communicate with another process on the same machine by transparently using LPC's, so RPC's could serve as a network wide communication mechanism. This RPC toolkit is an almost complete implementation of the Open Software Foundation Distributed Computing Environment (OSF/DCE). This means it is also compatible with the RPC's from other DCE-compatible environments.

To facilitate communication between a client application and a server application, a protocol between them should be defined as an interface. This definition is given in a language called the Interface Definition Language (IDL). Furthermore an application configuration file (ACF) must be generated that contains RPC attributes that do not directly relate to transmitted data between the client and the server. It mainly contains information for binding a client to a server. The IDL and ACF files can be compiled, resulting in stub functions that are used by both the client and server RPC applications. The client stub makes the application think it performs a local procedure call, and the server stub transports the request to the appropriate function. Results are sent back in the same way. Note that this mechanism is the same as discussed in chapter 6.

The header of the IDL file contains information needed to permit a client to locate, connect and utilize a server. Among this information is an unique identifier of the RPC interface, and an endpoint attribute. The endpoint attribute specifies the network transport protocol, and a protocol specific port used by the server to listen for client requests. The body contains function prototypes of the remote functions along with their parameter attributes. Because both client and server have to know the common interface, the user has to make sure that the IDL file is compiled at both client and server. The marshaling of parameters such as functions, pointers and arrays are discussed in more detail in [fin94].

The ACF file contains binding information, meaning information describing the logical connection between a client and a server. This binding can be handled automatically by the client and server stub code or by the application itself. If it is done automatically, it happens completely transparently behind the scenes for the user, and the user has no means of controlling which server is chosen. This however also means that there must exist a locator service that chooses an appropriate server. To my knowledge such a locator service is not standard available with Windows NT. The binding can also be performed explicitly by the client application itself, but then the remote processing is not location transparent. If the application could use a locator service mentioned before, the application could gain both location transparency and control over which server is the most appropriate.

As we have seen, the RPC toolkit can be used for distributing processes to remote machines. Location transparency is however not provided. Furthermore the distributed processing has to be managed by the user himself. Concurrency control, causal precedence preservation and getting a global view of the system all have to be implemented on top of Windows NT. The OSF Distributed Computing Environment, from which Microsoft has imitated their RPC interface, has been brought to Windows NT by the Digital Equipment Corporation [dec94]. Next we will examine if this could enhance Windows NT's remote processing ability.

15.1 OSF Distributed computing environment for Windows NT

The Open Software Foundation (OSF) is a cooperating group of companies trying to develop a standard Distributed Computing Environment (DCE). The OSF has reviewed many distributed system architectures, and has gathered the opinions of leading developers and researchers in the area of distributed systems [joh91]. This research has lead to a statement of which properties a DCE should posses. Using these a standard structure of the DCE has been defined, that should create a common API that can be used on top of existing systems. OSF DCE provides a set of integrated software services for use of distributed applications. This OSF DCE can now also be used as an additional API on top of Windows NT [dec94].

In OSF DCE the different machines are grouped in cells. One of the key components is the Cell Directory Service (CDS). It provides unique network addresses of named resources, such as users, servers and processes within the cell. This is done by using replicated name services, so that applications can use networked resources without concern of their physical location. Another defined service is the RPC, which is nearly the same as the Microsoft RPC including the IDL language. OSF DCE also uses the concept of threads.

Now the OSF DCE is also implemented for Windows NT. It provides Windows NT with DCE RPC's that can be transparently mapped to the Microsoft RPC. Applications written for the DCE can thus be ported to Windows NT. More important, also the CDS is implemented so it can be used by the DCE RPC for locating servers. There is also a name service interface daemon available, that allows native Microsoft RPC clients and servers to use the CDS the way DCE RPC does. The CDS on Windows NT provides a consistent mechanism for naming and locating users, applications, files etc. within a cell. The threads used within the DCE can be mapped directly to Windows NT threads. Applications that use Windows NT threads can continue to use them, but also new applications using DCE threads can be used.

DCE for Windows NT provides an OSF DCE conferment API, using a direct mapping to Microsoft RPC's and Windows NT threads. So although Windows NT does not offer total distributed computing, it could be 'plugged in' using for instance the DCE approach. Especially the CDS would be a valuable extension of Windows NT.

Suggested literature:

[fin94] - This document provides an detailed discussion about how RPC's must be implemented for Windows NT. It discusses the IDL and APC files, but also how parameters are passed from client to server and how they are marshaled.

[joh91] - This article is a very interesting discussion of the evolution of the OSF DCE. It first characterizes the properties a distributed computing environment should posses. These properties are generally the same as the ones discussed in Part I of this thesis. It then discusses how the distributed computing environment should be structured.

[dec94] - This is a background paper from the Microsoft Development News CD-ROM. It announces the introduction of OSF DCE for windows NT by the Digital Equipment corporation, and gives an short overview of the possibilities. Although the document may not be construed as a commitment by DEC, it still shows that Windows NT could be extended to a more distributed operating system by using the DCE API.

16. Windows NT file service

Windows NT supports three types of file services: the good old MS-DOS File Allocation Table (FAT) filesystem, the High Performance File System (HPFS) to support OS/2 and the New Technology File System (NTFS). We will only discuss NTFS because this is Windows NT's native filesystem. HPFS will not be used very widely on a NT system and the FAT filesystem is a bit too old. All filesystems use however the same drivers for automatic caching and disk striping discussed later. This is again possible due to the layered construction of the I/O manager.

NTFS provides a hierarchical name space on each volume. Internally file access is speeded up by using B-trees. A B-tree is a binary branching tree of linked data records and is reorganized each time a data record, or node, is added or deleted. The tree is reorganized so that it remains balanced, thus with an equal depth and equal number of branches descending from each level of the tree. NTFS does not provide a global namespace so that files can be accessed location transparent network wide. Here a name service should be used on top of Windows NT like for instance the global name service of OSF DCE mentioned in chapter 15.1.

NTFS is a recoverable fileservice, even though disk I/O including writes is automatically cached. NTFS maintains a log that keeps track of all operations that affect the structure of a volume. These operations are treated as atomic transactions against a database called the master file table. In case of a failure all not completed transactions can be rolled back using this log. In this way also the loss of not yet executed writes in the cache are dealt with.

Windows NT using NTFS can also provide some fault tolerance by using a form of replication. The two types of replication are called *striping* and *mirroring*. With disk striping data is divided into chunks. Each chunk of data is stored across the number of drives defined in a stripe set. In case of a data failure, the required data can be found at the other drives in the stripe set. Furthermore read access is better because data is distributed across multiple drives. By using mirroring a region of a disk, treated as a logical disk, can be backed up at another drive continuously. If the first drive fails, the second one can be used. This approach most resembles the primary backup method, while striping is more like an active replication method. The mirroring feature is only provided by the Windows NT advanced server. The advanced server can also extend disk striping with a parity check.

Suggested literature:

[ude92] - although this is mainly a product review of Windows NT, it discusses the file systems rather extensively.

[usi93] - in this very complete user guide for Windows NT the file systems are also discussed. It provides information as to how disk striping and mirroring are controlled in Windows NT.

17. Is Windows NT a distributed system?

Now we have come to answer the final question: “is Windows NT a distributed operating system?”. In order to say whether Windows NT may be called distributed or not, we will compare the characteristics of Windows NT with the working definition presented in chapter 4. Remember that this is called a working definition because it is no exact definition, it summarizes the characteristics that an ideal distributed computer system must have. If Windows NT complies to a major portion of these characteristics we will call it distributed.

As we have seen in chapter 14, Windows NT is designed with networking in mind. Windows NT standard provides peer to peer networking in workgroups, and by using Windows NT advanced server a network of connected NT workstations can be managed. The built in networking features are built to Microsoft LAN manager specifications, but also TCP/IP is supported. Due to the layered structure of the I/O manager other network protocols can be supported by writing the appropriate drivers. So the first two parts of the definition can be fulfilled: multiple computers interconnected by a network. Additionally, although we excluded this from our definition, within a single machine Windows NT supports symmetric multiprocessing.

Windows NT has also addressed the issues of heterogeneous hardware and extensibility. Heterogeneous hardware is dealt with by using a layered structure at the lower level of the operating system. A hardware abstraction layer is used for platform independence, while processor specific code is only placed in the micro kernel. Also the I/O manager contains some hardware specific device drivers at the lower layers which hide hardware specific issues from the rest of the operating system. At the highest level extension has been made possible by using the client/server model.

The third primary characteristic is that the multiple computers have to maintain some shared state. We divided this characteristic into multiple sub-properties. We will discuss the different subproperties separately.

System transparency:
Windows NT does not provide much system transparency to the user. Objects and resources can be accessed remotely using a hierarchical name space, but no global naming service is provided. This means that users have to know at which workstation the needed object is located, so they can perform binding to the object themselves. This lack of location transparency is a serious shortcoming of Windows NT. Using extensions like the OSF DCE API discussed in paragraph 15.1 could however be used to supply Windows NT with a global name service.

Distributed processing:
Windows NT uses the concepts of processes and threads. We have seen in chapter 13 that locally at a single machine the Windows NT executive supports concurrent processing. This is done by SMP in case of multiple processors and preemptive multitasking at individual processors. Thread and process synchronization is not done automatically, but the executive does provide programmers with synchronization objects that can be used with care for this purpose. Processes and thread are themselves handled as objects that become signaled when they terminate. Using this, processes and threads can be forced to wait for each other, so that causal relations can be created and managed at a single machine.

The distribution of processing across the network is not handled by the executive itself. For this purpose a RPC interface toolkit is standard included in Windows NT. This RPC can transparently be ported to the internally used LPC, so that RPC's can be used as a network wide communication mechanism. Problem again is that the binding of RPC client and server can not be performed location transparent without a global naming service. Concurrency control of the processes distributed across the network needs to be implemented by the user.

Distributed data:
We have discussed the native fileservice NTFS of Windows NT in chapter 16. Due to the lack of a global name service, the fileservice of Windows NT is also not location independent. Users can however access remote files if they possess the right privileges and know the name of server where it is located. Also a form of replication is provided by disk striping and mirroring. The user has to define himself which data has to be replicated to which location. Using the Windows NT advanced server, directories can also be replicated to user defined locations. These directories are updated at the primary server that sends the update to the other servers The directory may be read at every location. By using multiple advanced servers as domaincontrollers, management information can also be replicated using a primary backup protocol.

Access control:
A large amount of system resources and data can be shared by users in Windows NT. Examples of system resources that can be shared across the network are printers, cd-rom drives, tape drives etc. This is mainly achieved by using the object model. Different processes can open so called handles to objects, and thus effectively share the object. Also data like files and directories can be shared over a network. Access to data is serialized using special objects like mutexes, critical sections and semaphores. These objects in fact act the same as locks in locking schemes. If programmers want to control concurrent access to data they have to implement that themselves using these objects.

Let us summarize the result of our search for distributed properties. Windows NT has built in networking capability, so the first two primary characteristics multiple computers connected by a network are satisfied. Windows NT does however not provide much system transparency due to the lack of a global name service. Concurrent processing is supported at every individual machine using the concept of threads. Processes can also be distributed across the network by means of the implementation of RPC's. The executive only provides synchronization objects as tools for synchronizing local processes and threads. Network wide concurrency control of processes is not provided. Data can be distributed over the network and a user can configure Windows NT so that some replication management is provided. The global file service again lacks location transparency. Resources and data can be shared across the network. Special objects are again provided so that programmers can serialize access to data, and hence perform some transaction management.

From the above we can conclude that Windows NT itself is not yet a truly distributed operating system, although it possesses some of the required characteristics. Especially the lack of system transparency and transparent distributed processing leads to this conclusion. Windows NT also provides the user the tools to distribute data across the network, and even to replicate data. This however has to be done explicitly by the user himself, by designating data to be replicated to specific locations.

All hope is however not lost yet. The way in which Windows NT is structured opens up possibilities for the future. Windows NT uses the concept of threads that can be executed concurrently, and provides functionality to synchronize processes and threads. It also provides RPC as a means of communication to transport processes across the network, while internally a lightweight RPC mechanism is in use. The layered fashion in which the I\O manager is structured opens the possibility to enhance networking properties by writing new drivers, and even implementing new fileservices. In this way Windows NT could for instance be extended with a global file service.

Other distributed characteristics such as system transparency and networkwide distributed processing must be provided at higher levels of Windows NT. The client/server architecture at the user mode portion of Windows NT makes it possible to extend it with a subsystem that provides a distributed computing environment in together with low level drivers. That this is possible is proven by porting OSF DCE to Windows NT. Note that OSF DCE directly uses many mechanisms of Windows NT such as threads, LPC's and RPC's. This indicates that Windows NT probably is designed with future distributed computing in mind. For a widely commercially available operating system, Windows NT is already heading for the right direction, but a long road still lays ahead.

18. Literature

[BER87] Bershad, B.N.; Ching, D.T.; Lozowska, E.D.; Sanislo, J.; Schwartz, M.
A REMOTE PROCEDURE CALL FACILITY FOR INTERCONNECTING HETEROGENOUS COMPUTER SYSTEMS.
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 110-124; reprinted from: IEEE transactions on software engineering, vol. SE-13, no. 8, aug. 1987, pp. 880-894.

[BIR84] Birrell, A.D.; Nelson, B.D.
IMPLEMENTING REMOTE PROCEDURE CALLS.
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 89-109; reprinted from: ACM transactions on computer systems, vol. 2, no. 1, feb. 1984.

[CEL88] Cellary, W; Gelenbe, E.; Morzy, T.
CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS, Elsevier Science Publishers B.V., Amsterdam, 1988.

[COU94] Coulouris, G.F. and Dollimore, J. and Kindberg, T.
DISTRIBUTED SYSTEMS: CONCEPTS AND DESIGN., 2nd edition, Addison-Wesley pub., Amsterdam, 1994

[CUS92] Custer, H.
A GRAND TOUR OF WINDOWS NT: PORTABLE 32-BIT MULTIPROCESSING COMES TO WINDOWS., Microsoft Systems Journal, vol. 7, no. 4, jul.-aug. 1992

[DEC94] Digital Equipment Corporation
DIGITAL DISTRIBUTED COMPUTING ENVIRONMENT (DCE) FOR WINDOWS NT, Microsoft Development Network CD-ROM, no. 2, march 1994.

[ENS78] Enslow, Philip H.
WHAT IS A 'DISTRIBUTED' DATA PROCESSING SYSTEM?
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 5-13; reprinted from: Computer, january 1978, pp. 13-21

[FIN94] Finnegan, James
BUILDING WINDOWS NT-BASED CLIENT/SERVER APPLICATIONS USING REMOTE PROCEDURE CALLS., Microsoft systems journal, vol. 9, no. 10, oct. 1994, pp.65-79.

[GOS91] Goscinski, A.
DISTRIBUTED OPERATING SYSTEMS THE LOGICAL DESIGN., Addison-wesley, 1991.

[JOH94a] Johansen, D. and van Renesse, R.
SOFTWARE STRUCTURES FOR SUPPORTING DISTRIBUTED COMPUTING.
Brazier, F.M.T. and Johansen, D., "Distributed open systems", IEEE Computer society press, Los Alamitos, California, 1994, pp. 1-10

[JOH94b] Johansen, D. and van Renesse, R.
DISTRIBUTED SYSTEMS IN PERSPECTIVE.
Brazier, F.M.T. and Johansen, D., "Distributed open systems", IEEE Computer society press, Los Alamitos, California, 1994, pp. 175-179

[JOH91] Johnson, Brad Curtis
A DISTRIBUTED COMPUTING ENVIRONMENT FRAMEWORK: AN OSF PERSPECTIVE.
Brazier, F.M.T. and Johansen, D., "Distributed open systems", IEEE Computer society press, Los Alamitos, California, 1994, pp. 57-77.
Reprinted from:
Proceedings of EurOpen Spring '91 conference, 1991, Tromsø, Norway.

[LIN93] Linnell, D.
WINDOWS NT - CAN MICROSOFT MAKE THE JUMP FROM THE DESKTOP TO DISTRIBUTED COMPUTING? Data communications Int., Vol. 22, No. 6, april 1993, pp. 68-77

[MUL87] Mullender, Sape J.
DISTRIBUTED OPERATING SYSTEMS.
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 137-144; reprinted from: Computer Standards & Interfaces, Vol. 6, 1987, pp. 37-44

[MUL93] Mullender, Sape J.
DISTRIBUTED SYSTEMS, 2nd edition. Addison-Wesley pub., Amsterdam, 1993

[RIC93a] Richter, Jeffrey M.
MEMORY-MAPPED FILES IN WINDOWS NT SIMPLIFY FILE MANIPULATION AND DATA SHARING.,Microsofts systems journal, vol. 8, no. 4, april 1993.

[RIC93b] Richter, Jeffrey M.
CREATING, MANAGING, AND DESTROYING PROCESSES AND THREADS UNDER WINDOWS NT.,Microsofts systems journal, vol. 8, no. 7, july 1993.

[RIC93c] Richter, Jeffrey M.
SYNCHRONIZING WIN32 THREADS USING CRITICAL SECTIONS, SEMAPHORES, AND MUTEXES., Microsoft systems journal, vol. 8, no. 8, august 1993, pp.27-44.

[RIC94] Richter, Jeffrey M.
ADVANCED WINDOWS NT., Microsoft Press, Redmond, Washington, 1994.

[RUL94] Ruley, John D. (editor-in-large)
NETWORKING WINDOWS NT., Wiley, Chichester, 1994.

[STA84] Stancovic, John A.
A PERSPECTIVE ON DISTRIBUTED COMPUTER SYSTEMS.
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 28-42; reprinted from: Transactions on Computers, Vol. C-33, December 1984, pp. 1102-1114

[UDE92] Udell, Jon
WINDOWS NT UP CLOSE., Byte, oct. 1992, pp.167-178

[USI93] Contrib. authors: Branchek, B.; Eidson, R.; Kindel, C. [et al.]
USING WINDOWS NT., Que corporation, Cheltenham 1993.

[WIL87] Wilbur, Steve; Bacarisse, Ben
BUILDING DISTRIBUTED SYSTEMS WITH REMOTE PROCEDURE CALL.
Ananda, A.L. and Srinivasan B.,"Distributed computing systems: concepts & structures", IEEE Computer society press, Los Alamitos, California, 1991, pp. 77-88; reprinted from: Software engineering journal, sept. 1987, pp.148-159

[YAO92] Yao, Paul
AN INTRODUCTION TO WINDOWS NT MEMORY MANAGEMENT FUNDAMENTALS., Microsoft systems journal, vol. 7, no. 4, aug 1992, pp.41-49.

19. List of Figures